Agent Based Matchmaking and Clustering: A Decentralized Approach to Search

Ogston, E.F.Y.L.

2005

document version
Publisher's PDF, also known as Version of record

Elizabeth Ogston

Agent Based Matchmaking and Clustering

A Decentralized Approach to Search

VRIJE UNIVERSITEIT

Agent Based Matchmaking and Clustering: A Decentralized Approach to Search

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ter overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 5 april 2005 om 13.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Elizabeth Flora Yehlan Ogston

geboren te Oxford, Engeland

promotoren: prof.dr. F.M.T. Brazier
prof.dr.ir. M.R. van Steen

Table of Contents

Preface

In order to place the work that follows in context, this preface contains an informal discussion of the limitations of search methods used in current computer systems. The more serious reader may wish to skip immediately to the introduction of the research questions tackled in this thesis, provided in Chapter 1.

Computer systems are tools that are uniquely able to handle huge amounts of information. Accordingly, the ability to conduct search within large data sets plays an important role in many applications. In current computer networks, such as the Internet, however, the sheer volume of information contained can be overwhelming. Then again, in a sense this amount of data has always existed. The novelty in recent systems is our ability to access it, and more importantly to *find* the sliver of knowledge in which we are actually interested.

The World Wide Web in particular is remarkable for the ease with which anyone can make use of it to answer questions and to publish information. Its advantage as an information system is its flexibility: every user is able to individualize their view of, or presentation to, the Web to meet their own requirements. As it integrates into our daily environment, however, it is becoming apparent that the fundamental design of some commonly used web applications encroaches on individuals' control over the content they access.

Search on the Web is usually done by means of directories and search engines. In many instances these tools are able to return good results for users' queries, but there are also cases in which the range of searches they support is a limitation. For instance, say you saw a red silk cushion at a friend's house and wished to purchase one like it for your own living room. If you were to resort to the Yellow Pages, a typical directory, you would face the problem that your query "stores that sell cushions" does not match the information recorded in the directory. Thus you might be able to phone all stores listed under "interior decoration", but would not find the cushion if it actually came from an oriental gift shop. A search engine, which could look through the more detailed contents of online shops might make the search easier. On the other hand, it still requires that you somehow know the correct keywords to use. If the cushion was available online, but described as a "burgundy throw pillow, 100% tercel" you would still be unlikely to find it.

The range of queries answerable by directories and search engines can be expanded by recording more details. There is, however, eventually an upper bound on the amount of information, especially distributed dynamic information, that can be stored in a manner that allows it to be quickly searched. Thus there will always be some information that might be valuable to some searcher, but has been left out. Further, no intermediate computer system can store facts like the knowledge that if directly asked on a sunny afternoon after having

a couple of glasses of wine, your friend would actually be willing to give you his cushion. Such time and context-sensitive information can be ascertained only through a query made directly to the source of the relevant data, your friend.

Moreover, while search engines have the altruistic appearance of a desktop application that works for the user to answer queries, it should not be forgotten that the answers they provide are determined not only by user input, but also by the design of the data collection and search algorithms running on a remote server. These algorithms can be manipulated to suit purposes other than those of the searcher. Google bombing by bloggers is amusing, and the blocking of illegal sites by governments probably does more good than harm. On the other hand, ultimately the entity with the most control over the answers returned by any search engine is the organization that runs it. Similarly, by choosing a category structure the designer of a directory must make assumptions about which item characteristics will be most important to all future users.

These difficulties with search engines and directories, in which the system designer determines what information can be easily found, are embodied in the issues of "autonomy", the individual's right to self control. Autonomy is essentially a human notion, but the concept can be used to abstract from many technical issues that are often thorns in the side of designers of distributed systems. Private information, unknown information, inconsistent information, random behavior, component failures, outside attacks, changing conditions, updates and conflicts arising from a lack of coordination among multiple programmers can all be described as "things over which the system architect has no control." Designing algorithms for explicitly autonomous system components creates the possibility of mitigating the effects of these problems.

In this thesis we propose an algorithm that allows autonomous components to tackle two subclasses of content-based search; matchmaking and clustering. We divide data and queries among a large number of software "agents" which then implement search by means of peer-to-peer communication. While making peer-to-peer queries is far more inefficient, in terms of total work, than simply querying a directory or search engine, we show that the approach has some definite advantages compared to more traditional centralized algorithms. A central component requires more resources as the amount of data to be searched and number of requests increases. By removing such components we improve the scalability of the system, in terms of the cost incurred by any particular component. Additionally, since agents make queries directly to each other, the subject of queries is not limited by an intermediary's capacity, nor can replies be manipulated by a go-between. Agents thus obtain far more control over queries and replies, allowing them to flexibly tailor requests to the data source they are querying or adapt replies to fit the requester. This ability can be advantageous in many search problems. We demonstrate, for instance, how it can be used to improve the ability of clustering algorithms to find clusters in data sets with widely varying characteristics by allowing agents to choose cluster partners based on individually learned parameters in place of centrally dictated ones.

Chapter 1

Introduction

Content-based search refers to the process of finding items with some specified characteristics, as opposed to searching by a given label or "name." In essence it involves a separate examination of each individual candidate item. As the number of items grows content-based searching quickly poses performance problems. If, in addition, items are widely distributed across a network the cost of querying each item individually prohibits direct search.

Content-based searching in distributed data is therefore usually tackled via some method of consolidation. This can be done in one of two ways. The first option is to make a copy of all data items at a central location. Since this generally requires an excessive amount of storage space, and can cause a hassle keeping copies consistent, usually not the data items themselves, but summaries thereof are stored. These summaries can be organized into a directory structure to assist search. Alternatively, data can remain distributed and a commonly known scheme can be used to organize the (virtual) placement of items, creating a distributed directory which any user can follow to quickly narrow down an item's location. The Yellow Pages directory follows the first of these approaches, allowing users to find the location of a particular type of business by looking it up in a catalogue. The Dewey Decimal System takes the latter: a user searching for science books in a library need not consult a directory, he can simply walk to the "500" shelves knowing that paleobotany, for instance, is located at "561."

In computer systems central directories are far more common, but face a scalability problem both in the number of items they must store and process, and in the management involved in collecting items and keeping them up-to-date. Both the central directory and placement scheme approaches have the additional weakness that in creating summaries and deciding on index structure they must make a speculative guess about which characteristics will be important for *all* future searches. New solutions which extend the range of searches that can be made in very large, distributed, dynamic data sets are therefore of interest. The research question put forward in this thesis is thus:

> *Can a method of locating items, based on their content, be devised that avoids the expense and guess work of creating and maintaining a centralized directory or placement scheme?*

In exploring this question we follow an approach from the field of Multi-Agent Systems. This field advocates breaking up large complex systems into smaller interacting components, called agents. Agents are used for an extremely wide variety of purposes, from

modelling human behavior to selling units of electricity, and thus have correspondingly many definitions. For this work, however, the exact definition of an agent is unimportant. Rather, we are interested in three particular properties that are often embodied by the concept of an agent; (1) peer-to-peer communication, (2) autonomy, and (3) adaptivity. These three properties, in combination, greatly enhance the scalability and potential for distribution of systems in which they are incorporated. Unfortunately, maintaining these properties also places severe limitations on the methods by which search can be realized.

In peer-to-peer systems, system data and functionality are distributed over a large number of peer components. These peers communicate directly among themselves to access each other's portions of the data. Peers are limited in size and are also usually restricted to being of roughly equal importance. In general, tasks requiring a common view of the entire system cannot be delegated to a single component, but instead must be accomplished in a cooperative distributed manner. This restriction rules out the creation of a central directory peer and thus makes search difficult. It does, however, also have its advantages. The limitations on an individual peer's size make it more feasible to scale such systems, in some situations. In many cases simply adding a cheap peer to handle additional data is more cost effective than upgrading a central server.

The issue of autonomy relates to the ability of an onlooker to predict a component's reaction to a stimulus. Traditionally, components of a computer system work together because they are "told" to. A system designer, or user, plans their actions and reactions. Autonomous components, on the other hand, in concept do whatever they "want" to do. While decentralization, as epitomized by the peer-to-peer paradigm, is mainly concerned with the distribution of data and functionality, autonomy explicitly concerns the distribution of control. This is an important distinction when considering scalability issues. Large-scale systems are apparently hard to build because of limits on memory, processing, or communications resources. But, the real difficulty in designing very large systems comes not from the fact that they are big, but from the fact that they are uncontrollable. By distributing control among many domains, the internal adminstration of each domain can be kept manageable. By further minimizing the assumptions made within one domain about the behavior of other domains, the amount of knowledge needed for cooperation between domains, and the effects of failures or changes within one domain on others, can also be limited. For search in large-scale systems one way of exercising control is by means of a central directory. If control is centralized in this manner, however, queries are limited to public attributes that have been preselected by the directory. Queries that are instead made directly to the data source, leaving control distributed, can be over any characteristics the requester finds important. Moreover, the answers to direct queries can be tailored by the data source, depending on who is making them.

If the autonomy of system components increases, however, the flexibility of the interactions between them must also increase. When a component's behavior is precisely defined it is relatively easy to design another component to work with it. Reducing the assumptions that can be made about component behavior results in a larger range of possible interactions, and hence a much more complex interface. Therefore, the concept of autonomy goes hand in hand with the concept of adaptivity. Adaptive systems, while still controlled by a fixed program, adjust some aspects of their behavior to changing conditions, and thus can provide the flexibility required to support autonomy. Adaptivity in search allows for negotiation

processes, so that requesters and providers of data (or services) can revise requirements that could not originally be met, to match available replies.

A distributed setting, therefore, has a significant impact on the way in which search can be carried out. Distributing the location of data increases the time it takes to follow a directory structure when looking up items, but does not change the fundamental process. If, in addition, the central directory view of data, which dictates what characteristics are important and how data should be organized, is removed, a more challenging problem is created.

Search in general can exist in several forms. Specific queries such as "find *the* plumber Mr Jones", "find the *best* plumber" or "find *all* plumbers" ask for specific satisfying items out of an entire data set. The terms "all" and "best" obviously require a global overview of the data as any unseen data item could potentially change the answer to the query. Less obviously the term "the" also requires this global perspective since, though having found "Mr Jones" the other items in the data set become unimportant, the absence of "Mr Jones" can be determined only by examining every item.[1] The nonspecific query "find *a* plumber", in a data set where plumbers are not extremely rare, in contrast, can be satisfied without resorting to a global view. Similarly, a qualified nonspecific query like "find a good plumber" can be answered locally, providing that the qualifier is not overly restrictive.

In this thesis we study two forms of nonspecific search, matchmaking and clustering, which involve many simultaneous searchers. Matchmaking, from a coordinator's point of view, is the process of creating a pairing of a set of requests with a set of replies that meets some criteria. It has a number of properties that simplify search from the viewpoint of the individual requesters and providers. First, the search space is limited to only the currently available offers or requests. Second, it assumes that compromises will be made. Individuals must accept the partner they are given in the pairing rather then each demanding the partner which actually best suits them. Since neither the individual matches nor the overall pairing need to be optimal to be useful, matchmaking is well suited for a distributed approach to search.

Clustering is the process of dividing a set of items into groups, based on items' similarity [14]. This process consists of two parts, first determining which items are most similar to which others, and second, deciding on the boundaries between groups. The first part can be seen as the result of each item in a data set doing an individual search for other similar items. A traditional query can be implemented by creating a corresponding query data item, along with specifying some conditions that can be used to define the boundary of its cluster. Since items need not find specific matches, but can tune the measure of similarity they employ to accommodate whatever other items are available, a distributed approach can be followed. More interestingly, a clustering[2] formed by such a distributed process can be used as an estimation of the grouping that would have been created by a central directory. Thus, depending on how closely the clustering criteria match a later query criteria, the clustering structure can be used to expedite additional queries.

By demonstrating agent matchmaking and clustering algorithms that have the properties of peer-to-peer communication, agent autonomy, and agent adaptivity this thesis takes an

[1] Often an indexing system is used to avoid a full search in these cases, however an index in fact provides a global perspective since it requires that items are orderable and each item is considered during its creation.

[2] The term "a clustering" refers to a set of clusters found for a particular data set.

important step towards the overall research goal of implementing general search without the aid of centralized directory structures. The following chapters delve into a more focused version of the general research question: is it possible to effectively and efficiently match pairs, or group together sets of items, based on their content, without creating a centralized directory or placement scheme?

The account of the contributions made by this work is organized as follows:

- Chapter 2 presents an algorithmic framework in which agents are used to model the distributed search process. This algorithm is adapted in the following chapters to implement various versions of decentralized matchmaking and clustering. This chapter also describes and justifies some basic simplifying implementation decisions that are used throughout the work and describes the setup of the simulation environment used for testing the proposed matchmaking and clustering algorithms.

- Chapter 3 identifies the basic properties and limitations of a derivative decentralized algorithm for a straightforward matchmaking problem. It shows that matches can be found surprisingly efficiently, provided that the probability of an item randomly encountering a match is above a certain threshold.

- Chapter 4 demonstrates how the matchmaking procedure from Chapter 3 can be used to implement a traditional double auction, and introduces adaptive matching by means of agent bidding mechanisms.

- Chapter 5 makes the transition from matchmaking to clustering by exploring how well matchmaking agents cluster one-dimensional data. It further shows how the concept of local shared views can be used to improve clusterings.

- Chapter 6 investigates agent clustering in the traditional two-dimensional spatial data setting, and compares the clusters found by decentralized agents to those found by the centralized k-means algorithm.

- Chapter 7 demonstrates the use of these clustering agents in a high-dimensional text-clustering setting.

- Chapter 8 further explores agent adaptation and how it can be used to discover parameters normally required as inputs to clustering, such as the natural size of clusters. The resulting agent algorithm discovers clusters that compare favorably to those found by traditional centralized algorithms.

- Chapter 9 describes how the clustering agents developed in the previous chapters could possibly be used in future work to implement more general search using semi-clustered overlay networks, surveys how the questions that arise in this thesis relate to other research, and presents overall conclusions.

More detailed summaries of results can be found in the conclusions sections at the end of each chapter.

Chapter 2

Model and Notation

This work studies the search process through simulations of an abstract model of multiple agents making and answering queries. This section describes the choices made in defining this model and illustrates how the model can be used to create a framework search algorithm.

2.1 Design Goals and Rationale

Our research goal is to create a decentralized algorithm that shows an ability to find items in a large distributed data set. In order to achieve full decentralization, we choose to model such data sets as a collection of independent agents that represent not just the raw data, but instead wrap together data and search functionality. Each agent is thus made up of an item of data, along with query and/or reply functionality associated with that data. Accordingly, search becomes an attempt to discover a pairing between an agent with a query and another agent whose reply satisfies that query.

For example, to implement simple keyword-based queries in a document collection, each article would be represented by a single agent. These article agents would each have a reply function that indicates the presence or absence of a word in their text. Similarly, queries would be made by creating an agent that represents a set of keywords with a query function that tests if each keyword is present in a candidate article, as represented by an article agent. Allowing these article and query agents to intermingle in some way would result in a search process.

By considering more complex searches the advantage of placing data and search functionality together becomes apparent. More advanced queries can be made by simply modifying the query agents, for instance by including a knowledge of synonyms in their query functions. Likewise, article agents that contain meta-data about their text's actual subject could tailor their replies accordingly. In essence, the autonomy of agents in controlling the use of their data is maximized. When considering searches for heterogeneous services, rather then standard unchanging data, such individual customizability becomes essential.

Having chosen this agent representation for data and the testing of possible query-reply pairings, the challenge in creating a search algorithm lies in the definition of a process for generating candidate agent pairs to be assessed. We make the assumption that an underlying communication layer exists that allows any agent to contact any other agent for which it has an address. Agents are thus limited in their communications only by the amount of memory they have to store addresses for outgoing contacts, and the amount of time and resources

they have available to handle incoming contacts. Taking these limits to their extreme, and thus least demanding, case we stipulate that agents, at any one time, know of only a small number of "neighbor" agents. To restrict the cost of incoming communications we make this neighbor relationship symmetric: each agent is known by only a small number of neighbors. If this number of neighbors is bounded independent of the system size, this arrangement rules out the use of global coordinating agents, and thus maximizes decentralization.

In this setup, the only possible candidate replies to an agent query come from its neighbor agents. This means that in order to do an expanded search beyond their initial neighborhoods, agents must be able to acquire new neighbors. The only source of information on new addresses available to agents is their current set of neighbors. Therefore, we must allow agents to copy neighbor addresses from each other. Since learning about new neighbors requires forgetting about old ones, in order to keep neighborhood size constant, simply copying addresses can result in agents being forgotten all together. Thus, to reduce the chance of the network formed by the neighbor relations partitioning, we specify that instead of simply copying neighbor addresses, agents actually exchange neighbors with each other. This keeps both the number of neighbors each agent knows of, and the number of neighbors who know of each agent constant.

The efficiency of the resulting search process depends on which neighbors an agent queries and which neighbors are exchanged with which others. Managing this process optimally, however, can be costly. To keep agents simple we choose, at least initially, to make these decisions at random. This stipulation, along with the limitation on the number of neighbors of an agent, focuses our research on the minimal resources and abilities required by agents to realize useful search functionality. This line of reasoning results in a model, presented in the next subsection, which satisfies the following design goals:

- Maximize agent autonomy in determining if a candidate reply to a search is suitable.
- Maximize decentralization, thus maximizing scalability.
- Minimize the functionality and resources agents must possess in order to take part in the search process.

2.2 Basic Model

The model studied in this work consists of a set of N agents $A = \{a_1, \ldots, a_N\}$. Each agent a_i is characterized by an item of data, x_i, named its *attribute*. The agents' attributes together form a corresponding data set $X = \{x_1, \ldots, x_N\}$. Agents also have a small number, $\delta \ll N$, of *links* to other agents. Links represent bidirectional communication channels between their two adjacent agents, and thus define the *neighborhood* of each agent. Note that by considering bidirectional, rather than unidirectional, links we bound the resources an agent requires for both ingoing and outgoing communications.

Link ends are represented within an agent by a *port*. Each agent a_i thus has a set of δ ports $P_i = \{p_{i1}, \ldots, p_{i\delta}\}$. A port should be thought of as a peephole that provides, to an outsider, a particular view on its agent's data. In the search context, we consider each port to represent an *objective* of an agent for which it seeks a matching objective in another agent. At each moment in time an agent has, associated to each of its ports, $p_{im} \in P_i$, a link

which joins it to another agent's port, $p_{jn} \in P_j$. Thus a link, l, is made up of a pair of ports, $l = \langle p_{im}, p_{jn} \rangle$.

For each of its links an agent, a_i, uses a *matching function*, $m_i : P_i \times P_j \rightarrow [0, \infty)$, to determine its *similarity* to the *neighbor* agent to which the link joins it. The higher the value of m_i, the more agent a_i believes that its objective, represented by p_{im}, is met by that of p_{jn}. Agents, however, do not necessarily agree on their similarity to each other. That is, it is possible that $m_i(l) \neq m_j(l)$. In order to simplify the comparison of links to each other, we introduce a function, $s(l) = f(m_i(l), m_j(l))$, through which two agents can agree on a single *strength* for a link. This strength function could be, for instance, the maximum, minimum, or average of the two values, or, for more advanced agents, could represent a negotiation process.

Links have two properties that indicate the degree to which the agents they join are working together. First, a link can be either *nonmatching* or *matching*. This state indicates if the objectives of the two linked ports are compatible, or in other words represent an answer to the searches of the agents adjacent to the link. Matching links have adjacent agents with a high similarity, or rather, are links with a strength above some agreed threshold σ. Second, each link is either *nonconnecting* or *connecting*. This property indicates the degree to which the adjacent agents are cooperating in the search process. Two agents joined by a connecting link are able to swap neighbor agents with each other, thus modifying their link sets. Note that agents swap rather than copy links since this removes the possibility of an agent being forgotten, and ultimately of the link network partitioning, due to the limited number of neighbors maintained by each agent.

All told, agents can perform the following local actions:

- Swap neighbors over a connecting link, thus rearranging links.
- Upgrade a nonmatching link to a matching link, based on the link's strength and a current threshold, agreed upon by the adjacent agents.
- Similarly, downgrade a matching link to a nonmatching link.
- Upgrade a nonconnecting link to a connecting link according to some decision criteria based on available resources and the cost of swapping neighbors. We discuss this criteria in the next subsection.
- Similarly, downgrade a connecting link to a nonconnecting link.

2.3 Simplifying Specifications

By repeating the swap action agents can continually break unwanted links and form new ones, creating a process by which they search for matching links. Matching links indicate that agents have satisfied individual search objectives. In contrast, the set of connecting links, in essence, defines groups of agents that are able and willing to work together on the search task.

Together the agents and connecting links at any point in time form an undirected graph G, with vertex set A and an edge set containing all connecting links. We define each con-

nected component in G to be a *clique*[1], or cooperating group of agents. The agents are thus divided into a set of non-overlapping cliques, $C = \{c_1, \ldots, c_k\}$. Over a series of swaps an agent's clique defines which other agents' neighbors it could possibly become linked to. For simplification, we thus modify the model and specify that agents can exchange neighbors with any other agent within their clique, rather than only with directly connected neighbors.

The basic model leaves open the question of how agents select connecting links. In the absence of any concrete application-based criteria we choose to assume that matches between agents indicate an ability to work together more closely. We therefore couple the link matching and connecting properties by having agents choose connections on the basis of link strength. This gives two strength threshold, σ_{low} which defines the strength above which a link is considered to be matching, and σ_{high} which defines the minimum strength for a link to be connecting. Thresholds are agent variables, but conceptually we consider each link to have thresholds that are somehow agreed by the agents adjacent to the link. How this agreement is reached will be defined more specifically when we present the implementation of the various agent versions. We define $\sigma_{low} \leq \sigma_{high}$, reducing the possibilities for a link's type to:

Nonmatching (nonconnecting) From here on simply *nonmatching links*. Links whose adjacent agents have a low similarity.

Matching (nonconnecting) From here on *matching links* or *matches*. Links between similar agents, in the opinion of the adjacent agents.

Connecting (matching) From here on *connecting links* or *connections*. Links between highly similar agents that have agreed to work together in the search process by joining the same clique.

We take the view that matching links are links that an agent wishes to keep, and that when an agent chooses to downgrade a link it indicates that that link is no longer required. Thus we specify that only nonmatching links are exchanged in the swap process and that agents downgrade connecting links directly to nonmatching links, skipping the matching (nonconnecting) phase in between. We also make a distinction between decisions taken by agents and those made by a clique as a whole. While this differentiation is unnecessary in the general model, it becomes important for applications, like clustering, where the overall composition of agent groups makes a difference. This leaves us with the following modified list of agent actions:

swapping nonmatching links: Given two nonmatching links, $l_1 = \langle p_{cm}, p_{dn} \rangle$ and $l_2 = \langle p_{eq}, p_{fr} \rangle$, if the agents a_c and a_e adjacent to each of the links, respectively, are in the same clique, they are able to swap their links, producing $l'_1 = \langle p_{eq}, p_{dn} \rangle$ and $l'_2 = \langle p_{cm}, p_{fr} \rangle$. This action performed repeatedly among agents in a clique results in a permutation of the clique's nonmatching link set.

upgrading nonmatching links: *agents* can use their matching functions to evaluate nonmatching links. If the resulting strength is above some threshold σ_{low}, the two agents adjacent to the link can upgrade a nonmatching link to a matching link.

[1] Note that this is not the same as the usual graph theoretic definition in which a clique is a fully connected component in a graph.

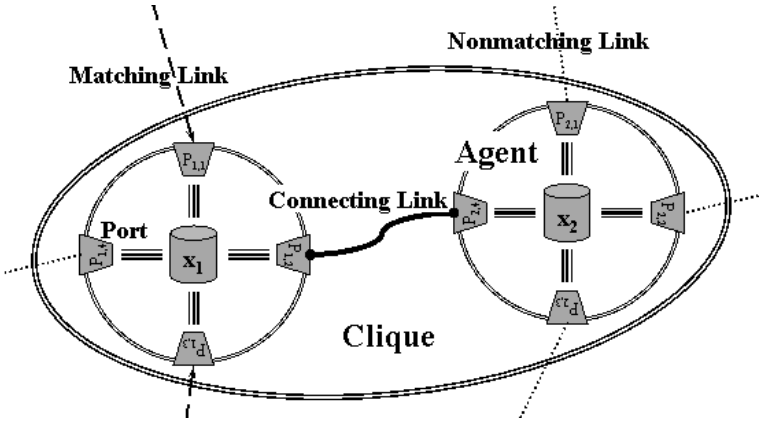


Figure 2.1: Diagram of model components

upgrading matching links: *cliques* can choose matching links to be upgraded to connecting links according to some threshold σ_{high}. This can result in two cliques agreeing to merge into a single clique.

breaking matching and connecting links: *cliques* can choose to downgrade a matching or a connecting link to a nonmatching link. This action can result in a clique splitting into two smaller cliques.

This modified version of the basic model, diagrammed in Figure 2.1, is used as the basis for the experiments presented in this work.

2.4 Implementation

Figure 2.2 provides a high-level pseudo-code specification of the algorithm underlying the model. The code depicts agents and cliques as distributed objects that communicate through remote procedure calls (RPC). The objects have the following components:

- **Process:** Object operations are achieved through a number of concurrent processes. Process methods are started when an object is initialized. Arguments in square brackets indicate that a number of processes are created, one for each argument.
- **External procedure:** An external procedure can be invoked through an RPC from outside the object. These procedures operate concurrently: each time a call is made a new process is started to run the procedure.
- **Internal procedure:** An internal procedure is available only from within the object itself.
- **Variables:** Variables are local data, visible only within an object. We follow the convention that data is implicity protected from concurrent access. Variables referring to other objects are in fact addresses. For simplicity's sake, however, we will use

```
1.  object Agent{
2.    acquaintances: set of [ neighbor:Agent, matching:Boolean, connecting:Boolean ]; //Agent's view of its adjacent links.
3.    c: Clique; //The clique of which this agent is a member
4.    process searchForMatches [for each aq in acquaintances]{ //One process for each acquaintance
5.      whenever ( ``now and then'' & matching = FALSE & connecting = FALSE ){
6.        next:Agent := c.tradeIn( aq.neighbor );
7.        aq:= [ next, ``negotiate match with next'', FALSE ];
8.    }
9.  }
10. virtual object Clique {
11.   members: set of Agent; //All agents that are members of this clique
12.   unwantedAcquaintances: set of Agent;
13.   external procedure tradeIn( a:Agent ) returns ( a':Agent ){
14.     unwantedAcquaintances.add( a );
15.     wait until( ``sufficient unwanted acquaintances have been received'' );
16.     return unwantedAcquaintances.remove( ``choose an unwanted acquaintance to return'' );
17.   }
18.   process searchForConnections{
19.     whenever (``good matches have been found'' ){
20.       nextLink:[ neighbor:Agent, matching:Boolean, FALSE] = ``choose an unconnected link'';
22.       if ( members.contains( nextLink.neighbor ) ){
23.         ``upgrade nextLink into a connection'';
24.       }else if ( bestMatch.neighbor.c.requestConnection( self ) = ACCEPT ){
25.         ``upgrade nextLink into a connection'';
26.       }
27.     }
28.   }
29.   external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when (``available''){
30.     if (``proposed clique OK''){
31.       return ACCEPT;
32.     }else return REFUSE;
33.   }
34.   process reconsiderCompositon(){
35.     whenever (``current composition may be improvable''){
36.       nextMatch:[ neighbor:Agent, matching:FALSE, connecting:Boolean ] = ``choose a matching link'';
37.       nextConn:[ neighbor:Agent, matching:Boolean, connecting:FALSE ] = ``choose a connecting link'';
38.       if ( nextMatch != null ){ ``downgrade nextMatch to a nonmatching link''; }
39.       if ( nextConn != null ){
40.         ``downgrade nextConn to a nonConnecting link'';
41.         if (``clique is no longer connected''){``split off a new clique'';}
42.       }
43.     }
44.   }
45. }
```

Figure 2.2: High level agent and clique implementation.

direct references to variables within other objects rather than explicitly showing every get and set operation. Also, in some cases we would like to indicate that an object referred to by a variable is co-located with the parent object, forming a single actual object. For these variables we will use the tag **internal** (for instance in Figure 3.1). We will allow co-located objects to access each other's internal procedures.

The agent paradigm has some further peculiarities beyond the standard distributed programming conventions. First, an emphasis is placed on the fact that agent actions can be initiated either by an external request or by some internal trigger. We indicate this idea of "autonomous action" by means of the **whenever** keyword which states that a process starts a given action each time the given condition is met. Further, agents are considered free to choose whether or not to act upon external requests. We indicate this by means of the **when** keyword which gives a condition for acceptance of requests. Finally, in a full agent model calls can be lost, or even misinterpreted. For simplicity's sake our implementation assumes perfect messaging with a notification of ignored calls, though we do make account for lost calls in our final design.

The **virtual** tag on the clique object in Figure 2.2 indicates that cliques exist only in concept, not as actual entities. Clique operations must in fact be achieved through cooperation among a number of distributed member agents. Some of these actions, however, require information about the clique's members as a whole, indicated by the set notation *.contains* (also *.all* and **for each**, used later). Others, like upgrading and downgrading links, require informing members affected by the action. In consequence, the actual implementation of a clique must involve a fairly large amount of communication between its members. This communication load becomes a limiting factor in the number of agents a clique may contain, as discussed in Section 3.4. Our actual implementation of clique operations uses a simple coordination mechanism; we specify that in each clique one agent is elected to act as the "coordinator" of the clique and maintains a clique map with the information needed to carry out joint operations. This map includes not only the addresses of the agent members of the clique, as shown explicitly in the pseudo-code, but also the state of each of the clique links. Member agents must inform the clique coordinator each time information in this map changes.

With this in mind, the details of the code can be explained as follows. Agents each have a set of acquaintances (line 2) which represent the model's port concept. Agents further have a process, *searchForMatches* (lines 4-8), which is invoked once for each acquaintance, as indicated by its argument. This process implements the swapping of unmatched links with other clique members and determines if new links should become matched. The process's **whenever** statement (line 5) continually checks an acquaintance's status, until it becomes nonmatching. It then requests from the clique a new neighbor for the link, by means of the clique's *tradeIn* procedure (line 6). The clique eventually returns a new neighbor agent, and the agent then resets the values of the acquaintance, calling procedures in the new neighbor agent to determine the new link's matching status (line 7). Note that there is a separate *searchForMatches* process for each acquaintance, thus the call to the clique *tradeIn* procedure does not block other agent operations. Also, the clique object *tradeIn* (lines 13-17) is an external procedure, meaning that each invocation is handled by a separate process.

The cliques play a coordinating role in the swapping routine through the *tradeIn* procedure. This procedure collects unwanted link ends, shuffles them in some way and then returns them to waiting agents. In the model this is described as a permutation of the link ends. A true permutation could be achieved if "sufficient" in line 15 was taken to mean all nonmatching links in the clique. Over time the mixing behavior results in agents finding possible matching links among their clique's neighbor agents.

The aim of the cliques is to choose some links from among the matching links found by the agents to form connections, thus mutating the clique composition. This is done by the process *searchForConnections* (lines 18-28) which creates new connections, and the procedure *reconsiderComposition()* (lines 34-44) which picks unprofitable matches and connections to downgrade. Both processes are triggered by **whenever** clauses which, in the absence of perfect information, require the clique to make some heuristic judgement about its current composition. These heuristics depend on the search task we hope to achieve and thus are filled in later. When a clique wishes to upgrade a matching link to a connection it negotiates the upgrade with the clique on the other end of the link by means of the *requestConnection* procedure (lines 29-33). Since creating a connection results in two cliques merging into a single clique, this procedure is protected by the clause **when** ``available''. This allows for cliques rejecting merges simply because they are busy doing something else, for instance merging with another clique.

In general, we take a loose approach to synchronization between objects. At times, agents or cliques would like to know that a certain global condition has been met, yet as local entities have no means of checking. Instead we design the code to work on the principle that over time certain assertions, as represented by the **whenever** clauses, are likely to become true. This implies that agents merely have to wait for a time, after which they can simply start to act. The underlying assumption is that by the time they take actions the desired conditions will prevail. If by some chance the conditions do not hold and a mistake is made, the algorithm is designed in such a way that corrective actions are naturally taken. In other words, from a correctness point of view no harm is done by taking premature actions, they will only affect performance.

2.5 Simulation Details

In this thesis we discuss two basic instantiations of the abstract model, along with variations of each. In the matchmaking models, we consider each port to be an abstract task for which its agent wishes to find a match, according to some abstract matching criteria. Cliques are used merely as a way for agents to obtain new links. In these experiments our main concern is to determine if, and how quickly, agents can find matches. In the clustering models cliques play a more prominent role, representing explicitly sought groups of agents, while ports are simply used to identify potential group members. In these experiments we are more interested in the properties of the groupings that are formed than in the actual links that are established between individual agents.

The results presented in this thesis were obtained using two different simulator implementations. The work on matchmaking is done using a straightforward centralized clock to regulate agent actions. In each time interval the agents being simulated are given the chance to act, one at a time, in a given order. Calls to other agents are implemented as procedure calls, and thus result in immediate action. The work on clustering, on the other hand, uses

a more advanced simulation model in which calls between agents are implemented using message passing. Messages are placed in a queue, and dealt with one at a time, in the order in which they are generated. Thus calls to procedures in other agents could take some time to realize actions and to return results, via a return message. All agent actions are triggered by calls from other agents, instead of by a clock. This second simulation model produces more realistic distributed behavior since it allows for a time delay between a requests being made by one agent and the request being acted upon by another. It also allows us to check that the system behaviors observed are not due to the synchronizing effect of a clock signal. We did use this second simulation to confirm the matchmaking experiments in Chapter 3, though we do not report these experiments. We are further able to run the second simulator in emulation mode, placing sets of agents on separate machines, to examine the effect of varying message delays between different agents. However, this work is still in progress, and thus not reported in this thesis.

2.5.1 Timing

The experiments presented in this work are all simulations run on a single processor. True timing for a distributed agent system can thus only be estimated. We use two versions of timing in the course of this work, initially a simple centralized clock that keeps agent actions strictly synchronized, and later, a more realistic message-triggered synchronization. Both versions measure time in turns. Where clarity is needed turns in the centralized time model are labelled $turn_1$ and those in the weak synchronization model are labelled $turn_2$.

Centralized Time

For the centralized clock model, time is divided into *turns*, each of which consist of four phases. During each phase each agent is given the chance to perform a particular action. In the work presented in Chapters 3 and 4 we use a fixed order, though in [32] we also considered randomizing the order in which agents act and found this made no significant difference. The clock phases are:

Phase 1 (Connecting): Agents attempt to upgrade matching links into connections. The **whenever** clause in *searchForConnections* is triggered.

Phase 2 (Mixing): Cliques return new neighbors to their agents. The **wait until** clause in *tradeIn* is triggered. Since all agents will have called *tradeIn* for all of their nonmatching links during the previous Matching phase, a full permutation of all nonmatching links in the clique is achieved.

Phase 3 (Matching): Agents test all of their nonmatching links, upgrade good ones to matches, and return unwanted neighbors to their clique. The **whenever** clause in *searchForMatches* is triggered.

Phase 4 (Breaking): Cliques choose some of their matching and connecting links to be downgraded to nonmatching links. If, as a result, a clique is no longer joined by connecting links, it splits into smaller cliques. The **whenever** clause in *reconsiderComposition* is triggered.

Weak Synchronization

In a real distributed agent system it cannot be assumed that agents will all act at the same speed. Concrete agents could require vastly different amounts of computation time to decide matches or run on different speed processors, and bandwidth between agents could vary dramatically. To more accurately reflect this, in the later clustering experiments we remove the central clock signal and instead base actions purely on the messages received from other agents. Since in the experiments agents all run on the same processor, they still run at roughly the same speed, though we have done some initial distributed experiments that indicate that this form of timing is viable.

For instance, the **wait until** clause in the clique *tradeIn* procedure can be triggered each time all of a clique's currently unmatched neighbors have be received. This condition synchronizes the speed at which agents within a clique search and synchronizes a clique's actions with its neighbors. The heuristics used for the other **whenever** clauses are described in Chapter 8.

For comparison, time in this model is also measured in *turns*, one turn consisting of $100 \times N$ messages, where N is the number of agents. Given the fact that all agents handle roughly the same number of messages, this provides an arbitrary time interval somewhat analogous to that in the centralized model where each agent performs a set of actions each turn, which allows us to compare the estimated running time of systems with varying numbers of agents. Since agent operations are simple, and in later experiments clique operations are also bounded, this estimation of running time reflects the assumptions that communication will be the largest cost in the system, and that agents act relatively independently of each other.

2.5.2 Initialization

Where initial links come from is a bootstrapping problem, we assume they are derived from the placement of agents in a network or some other application-dependent source. In our experiments we use an initial setup in which agents are given completely random links, all of which are nonmatching.

We further only consider scenarios where the set of agents is static, and thus do not define how agents can be added or removed from the system. Such an effect can be achieved, however, by allowing agents to know of more addresses than they have ports, thus allowing a new agent to simply copy or take over initial addresses from an existing agent. An agent that exits the system in a planned manner can simply pass on its addresses to one of its neighbors. Agents exiting unexpectedly can result in other agents being forgotten, though this effect can be minimized if agents each keep track of how many other agents know about them, and inject new copies of their address as necessary.

Chapter 3

Matchmaking

In this chapter we study a straightforward matchmaking problem in which agents each have a number of tasks for which they seek matching tasks in other agents. We develop a version of our general model to represent this problem, show that agents can produce good solutions for such matchmaking problems without needing to resort to a coordinator, and explore the conditions under which this is possible. The work in this chapter was originally presented in [30], [31], and [32].

3.1 Introduction and Related Work

Matchmaking is the process of creating a pairing between two sets of objects, according to some criteria. From the point of view of each object, matchmaking is equivalent to a search for a member of the other set that meets certain requirements. The optimization criteria for matchmaking, however, are usually taken from the global perspective. For instance, the matching is sought in which the most pairs are formed, or some other overall criteria is optimized. Therefore, the most desirable pairing, while acceptable to the majority of objects, does not necessarily result in an individual object's search request being satisfied in the best way possible. In this chapter we investigate matchmaking in the setting, in which it is often studied, of agents performing joint tasks. In terms of agents, matchmaking is portrayed as a number of consumer agents with requests for services attempting to find partners among a number of provider agents offering services. In the literature a number of possible solutions have been proposed; using facilitators, employing market mechanisms, exchanging recommendations, and forming coalitions.

The most straightforward approach to matchmaking is to use a facilitator, such as a broker, to centrally determine matches [22]. There are a number of different versions of facilitators, differing in whether consumer or provider information is stored, and if the facilitator directly connects consumers and providers or acts as an intermediary for transactions [6]. However, all facilitator architectures involve summaries of offered services or requests being stored in a central location. Agents wishing to find matches then apply to this location.

Facilitators trade off the cost of storing and maintaining a central directory for the ease with which good matches can be found. Their central nature means they can become a system bottleneck, though this problem can be lessened through distribution. Mullender and Vitányi [24] present a general model of a distributed directory service and study its

memory and messaging costs. Jha et al [17] discuss splitting a single facilitator's function among a number of agents.

Communication cost is not the only disadvantage of using a facilitator. A further problem is created by the fact that storing offers and requests centrally means that a language must exist in which they can be described. These descriptions must on the one hand be compact and easily comparable, and on the other hand flexible enough to cover all the situations the facilitator may need to handle. Thus, studies of facilitator architectures often go hand in hand with the study of ontologies. LARKS [47], for instance, provides a possible representation scheme that can enable the process of choosing matches by defining a structure for requests and advertisements of services within a given ontology. GRAPPA [7] also provides a structure for describing requests to make complex objects more easily comparable. The ontology problem, however, is far from easy to solve. Requiring that requests and offers are expressed in any particular language or form automatically restricts the domain over which matches can be made. The best that can be done is to make a wise choice of language, but for advanced domains that require a great deal of flexibility, as is generally the case when components are allowed to be autonomous, such languages are often not descriptive enough [35] . Even human languages are insufficient to make fully precise advertisements in many areas, as anyone who has ever written a job application has probably discovered. Thus while facilitators can often suggest possible answers to a search, they also often miss suggesting many other, perhaps better, answers due to incomplete information.

One alternative to using facilitators for matchmaking is to use commodity[1] markets to exchange services. Markets involve a set of requesters and providers making successive bids and offers for a particular good, and updating their requirements based on the requirements of others. Markets allow a great deal of flexibility in certain directions, usually the price of a service, in exchange for the fact that all services offered in the market must be roughly interchangeable. Markets for task distribution among agents were originally studied by Smith in Contract Net [43]. There is also a significant amount of other research in the field of Economics dedicated to market mechanisms and how they can be used to fairly distribute goods. This is easily extendable to trading services in place of goods [50].

The market mechanism, however, depends on a marketplace, which is often centralized. In general, all participants in agent markets are modelled as knowing all bids and offers being made, corresponding to the most understood case where participants have complete information about current prices. Distributing this price information to all agents must be done by means of repeated, expensive, broadcast operations. The central market controller must keep a list of all the participating members, and each bid or offer can involve sending a message to each participant. This can result in a prohibitive amount of traffic in large agent markets since usually all participants frequently update their bids and offers.

Matchmaking can also be attempted in a more distributed manner by having requesters and providers locate potential partners by asking each other for recommendations. In acquaintance networks [10] [44] [53] agents in a fixed network ask neighbors to pass on requests for services. For this, each agent needs to know only who its neighbors are. In terms of the number of communications between two agents, however, this can be even more costly than having a single agent broadcast requests directly to all other agents, since agents

[1] Commodity markets consider only a single type of item, or *good*, being exchanged. Systems that exchange multiple types of goods can be modelled as multiple concurrent commodity markets.

can receive the same message multiple times. In addition, it requires limiting the distance a message can travel, or having agents remember which messages they have seen, to prevent messages being passed round in circles.

An alternative way of using acquaintances to allocate tasks is to form coalitions [21]. Coalitions are groups of agents with common interests, that make an agreement to work together. Since matches to a request for a service are likely to be found in an agent's own group, coalitions in effect partition and thus reduce an agent's search space. Then again, choosing which coalitions to form can be difficult. Forner [11] discusses using recommendations among agents that naturally group into categories to find other like-minded agents. Shehory and Kraus [45] present a distributed algorithm for coalition formation. It, however, involves recursively calculating possible coalition values, and picking the coalition with the best system wide value. Coalition values can be calculated in parallel, but this phase requires each agent to know of all other agents in the system. Moreover, to determine the best value they resort to broadcast.

In this chapter we use our general model to implement an alternative decentralized form of agent matchmaking and explore its effectiveness, scalability and applicability. We study an abstract scenario in which we ask if, and how quickly, finding a good pairing of tasks is possible, given a varying availability of possible matches for each task. Since the most important aspect in this decentralized scenario is how common matching tasks are, we leave aside the question of how matches are actually determined, representing this part of the matchmaking problem through abstract matching functions. In Chapter 4 we demonstrate how this type of matchmaking can be used to implement a more concrete application, a fully decentralized agent auction. We leave the more involved question of forming clusters or coalitions to later chapters.

3.2 Model

The general agents, described in Chapter 2, are used as the basis for the experimental model for this chapter. In order to represent the matchmaking problem we make a number of further specifications. The objectives of an agent, represented by ports, are embodied as tasks which the agent would like to perform jointly with some other agent. Agents thus each have a number of tasks for which they wish to find a matching task. The overall goal of the system is to create as many matching task pairs as possible.

We consider abstract tasks, which are each of a particular type, chosen out of the set of categories $T = \{t_1, \ldots, t_M\}$. One task is assigned to each of the δ ports of each agent. Agents are thus characterized by a set of δ tasks. An agent's task set can be thought of as the attribute that describes it. Conceptually, the tasks assigned to each agent somehow fit with its overall purpose. To keep things manageable, however, in the experiments in this chapter the type of each task is simply chosen uniformly at random out of the entire category space. This results in all tasks being equally easy to find, thus most clearly showing the effect of varying the task range, M.

To assign a strength to a link between two ports, p_i and p_j, we use a matching function that is a function of the task categories of the ports, t_i and t_j respectively:

$$m(t_i,t_j) = \begin{cases} 1, & \text{if } (t_i,t_j) \text{ is a matching pair} \\ 0, & \text{otherwise.} \end{cases}$$

Each category has one, and only one, match. Matches are symmetric and all agents use the same matching function, making agreement on a link's strength automatic. That is, for a link l between agents a_i and a_j, $s(l) = m_i(l) = m_j(l)$. Agents also all use the same fixed thresholds when determining a link's matching and connecting status. σ_{low} = 1, thus all links with matching categories become matches. σ_{high} = 1 thus the *searchForConnections* procedure upgrades all matching links to connections. Since we consider the ports of each agent to be independent tasks, and since we found it makes little difference in these experiments, we keep things simple and do not prevent two ports of the same agent from matching to each other.[2]

Figure 3.1 gives a pseudo-code description of this matchmaking procedure, which can be compared to Figure 2.2 to see how it implements the general model. All our matchmaking experiments are run using the centralized time model. Therefore we use the clock phases, described in detail in Section 2.5.1, to trigger process actions. During a turn each agent receives a clock signal to mark four clock phases in which it preforms a corresponding action: (1) Connecting, (2) Mixing, (3) Matching, and (4) Breaking. For instance, as before, link swapping is achieved through the Agent *searchForMatches* procedure. This is triggered by phase 3 of the clock (line 19), replacing the general "now and then" specification in the general model. New neighbors are then returned by the Clique in clock phase 2 of the next turn (line 29). As before, agents have a set of acquaintances that represent their links (line 16), but we now define separate objects to represent ports and tasks. Tasks, however, are actually located within ports, and ports within agents, as indicated by the **internal** keyword (in lines 2 and 16). Therefore, in line 21 an agent can determine the matching status of a new link by using the Task *matches* procedure. We use the integers, modulo M, to indicate task categories and pair them off (1,2), (3,4), etc. to define matching pairs.

Similarly, upgrading matching links to connections is still accomplished through the Clique *searchForConnections* procedure, which is now triggered by clock phase 1 (line 33). Initially there is no limit on clique growth, thus *requestConnection* always returns OK (line 45). Matches and connections are never downgraded in this model, thus *reconsiderCompostion*, triggered by clock phase 4, does nothing (line 49).

3.3 General Behavior

The agents in the above model will be successful at matchmaking if, during an experimental run, the majority of links become matching links. Since all matches become connections, large cliques of agents would result. In the this section we investigate the basic behavior of our matchmaking model, asking if and how cliques form as we vary the values of M, N, and δ. In Section 3.4 we examine the effect of placing limits on which agents may join a clique, and in Section 3.5 we investigate the effect of more complex matching behaviors and dynamic agent task sets.

[2] Later, in the clustering experiments where ports of the same agent are more strongly related to each other, we do make this restriction.

```
1.  object Port{
2.    task: internal Task; //This port's task to be matched
3.    neighbor: Port; //Port in a neighbor agent to which this port is linked
4.    matching: Boolean; //Match status of the link
5.    connecting: Boolean; //Connect status of the link
6.    myAgent: Agent; //The agent this port is a member of
7.  }
8.  object Task{
9.    type: Integer; //Chosen out of [1 ... M]
10.   internal procedure matches( t2:Task ) returns ( Boolean ){
11.     if ( ``type is odd'' )return ( (type+1) mod M = t2.type);
12.     else return ((type-1) mod M = t2.type);
13.   }
14. }
15. object Agent{
16.   acquaintances: internal set of Port; //δ links per agent
17.   c: Clique; //The clique of which this agent is a member
18.   process searchForMatches [for each aq in acquaintances]{
19.     whenever ( ``clock phase 3'' & aq.matching = FALSE & aq.connecting = FALSE ){
20.       aq.neighbor := c.tradeIn( aq.neighbor );
21.       aq.matching := aq.task.matches( aq.neighbor.task )
22.   }
23. }
24. virtual object Clique {
25.   members: set of Agent; //All agents that are members of this clique
26.   unwantedAcquaintances: set of Port;
27.   external procedure tradeIn( p:Port ) returns ( p':Port ){
28.     unwantedAcquaintances.add( p );
29.     wait until( ``clock phase 2'' );
30.     return unwantedAcquaintances.remove( ``random element'' );
31.   }
32.   process searchForConnections [for each m in members]{
33.     whenever ( ``clock phase 1'' ){
34.       for each aq in m.acquaintances {
35.         if ( aq.matching = TRUE & aq.connecting = FALSE ){
36.           if ( aq.neighbor.myAgent.c.requestConnection( self ) = ACCEPT ){
37.             aq.connecting := TRUE;
38.             ``join the cliques'';
39.           }
40.         }
41.       }
42.     }
43.   }
44.   external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when ( ``available'' ) {
45.     return ACCEPT;
46.   }
47.   process reconsiderCompositon(){
48.     whenever ( ``clock phase 4''){
49.       ``do nothing'';
50.     }
51.   }
52. }
```

Figure 3.1: Pseudo code for matchmaking version of the agents.

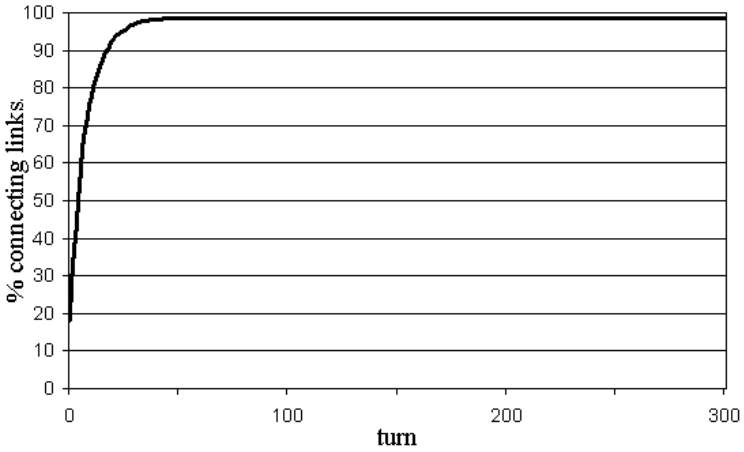


Figure 3.2: Percentage of connecting links over time; $M = 10$, $\delta = 3$, $N = 2000$.

3.3.1 Clique Formation

One factor in determining the ease with which agents find matches should be the number of ports, δ, which defines the size of the neighborhood of each agent. We first investigate systems of agents with three ports each. Three ports should be the lowest interesting number, agents with one port can at best form pairs, and agents with two ports can only form chains and rings, neither creating large cliques.

We first consider an extremely low number of categories of tasks. Figure 3.2 shows a trial run with 10 categories and 2000 agents. The y-axis shows the percentage of ports that have found matches, and the x-axis shows time, measured in turns. Figure 3.2 shows that with this low number of categories almost all the links quickly become connections. We do not expect 100 percent of the ports to have found matches at the end of the trial since we assign categories at random, and thus there can be an odd number of ports of a particular category.

Figure 3.3 shows the other extreme, a trial with 300 categories of tasks, again for 2000 agents. Here the chance of ports finding a match is so low that only a few tiny cliques form. Some agents have matching agents placed in their neighborhood by the initial random setup. However, after these matches are made, the agents find themselves surrounded by others with whom they have no tasks in common, and no further connections are formed.

We now consider a system with an intermediate number of categories of tasks. Figure 3.4 shows a trial with 80 categories. We again create 2000 agents, giving an average of 75 ports of each category, enough that the chance of having a very uneven distribution of categories is low. Intuitively we expect that some agents will find matches in their initial neighborhood, thus creating many small initial cliques. In each turn some additional agents will find matches so that these cliques will grow. Two randomly chosen ports have approximately a 1/80 chance of matching, so cliques will probably grow steadily and slowly, and eventually reach a point where the available search space for each clique contains no more matches. At this point, the system will stop changing.

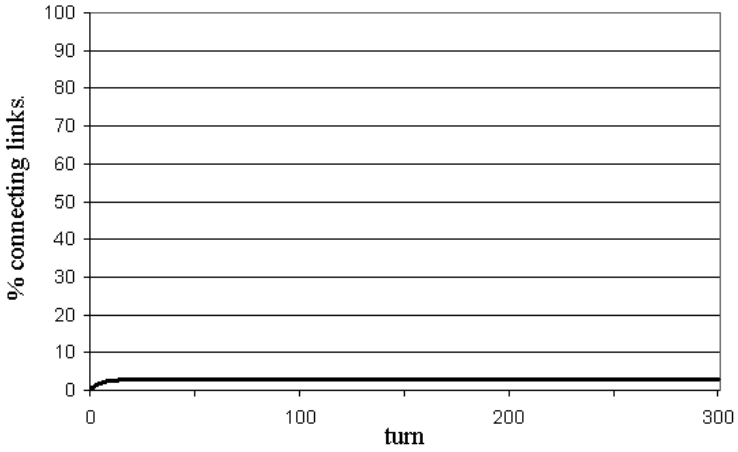


Figure 3.3: Percentage of connecting links over time; $M = 300$, $\delta = 3$, $N = 2000$.

This intuition, however, fails to materialize. There are a couple of surprises in Figure 3.4; first the fact that 95% of the ports find matches, and second, the shape of the curve. After a set of initial connections, matches are found slowly as we expected, but then the rate at which matches are found increases and the remaining ports are matched much faster than we expected. This behavior is consistent over multiple random starting conditions, as we find in the 100 trials studied in Section 3.3.2. The turning point in the rate at which matches are made varies slightly but not by much.

A further inspection into the size of the cliques on each turn gives a better intuition into this behavior. This is hard to depict in a graph, so instead we give a description of our observations. The initial random setup places some agents in the neighborhood of others to which they match. These agents join together, forming many small cliques. These initial cliques account for the small number of connecting links formed at the start of each simulation. Since agents can exchange nonmatching links with any other agent in their clique, joining one of these cliques broadens an agent's neighborhood. In Figure 3.3, for which there were a large number of categories, the chance of two tasks matching is so low that agents fail to find any further task matches, and thus no more connections are formed. In Figure 3.4, on the other hand, the probability of a port finding a match is high enough that some of these cliques grow slowly, resulting in the first part of the curve, labelled region A. Once one of these cliques becomes large it will have many unmatched ports, that is ports adjacent to a nonmatching link. The search space of each of these ports, and thus their chance of their link forming a connection, grows. Moreover, when a new connection is formed the chance that it is to another clique instead of to an individual agent becomes large. Once this happens the clique grows even bigger, and its chances of connecting to other cliques are even larger. In regions B and C of Figure 3.4 the system acts more like Figure 3.2, the 10 categories case. In region B, there is a phase where all the small cliques are rapidly grouped together into one large clique. This clique has many unmatched ports, which quickly form connections to the remaining free agents in the system. The rate at which connections form then slows;

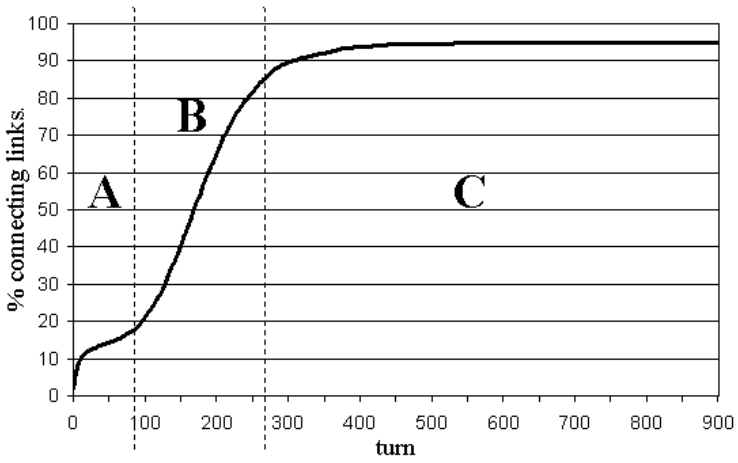


Figure 3.4: Percentage of connecting links over time; $M = 80$, $\delta = 3$, $N = 2000$.

most new connections are between agents already within the clique. In region C of Figure 3.4, the agents are mostly interconnected and the search space for each unmatched port is all of the other unmatched ports. Thus, in this region, we see a slowing of the connection rate as the remaining unmatched ports in the final clique pair up.

3.3.2 Number of Task Categories

In the previous section we saw that the matchmaking system has two basic kinds of behavior; with 300 categories of tasks the agents remain separate and almost no matching links are formed, while with 10 categories agents connect into a single large clique with almost all ports finding matches. How this behavior changes between these points is of interest. We examined one point, 80 categories, between these two extremes and found it acted most like the 10 categories case. We would like to know more precisely how this behavior changes as the number of categories is increased from 10 to 300. Possibly the percentage of connecting links at the end of the trial slowly diminishes. Alternatively, there may be a region where some trials form a large clique as in Figure 3.2 and some behave as in Figure 3.3. Overall, we would like to know the maximum value of M that the system *supports*. That is the maximum value of M for which large cliques are formed.

In further experimentation we in fact find that in all trials, no matter what the parameters, either the agents form more than 90% of possible connections, or do not develop cliques at all, finding less than 15% of possible connections. We will call the successful trials, in which a large clique is formed, *connecting* trials. Figure 3.5 shows the behavior as the number of categories changes. The x-axis shows the number of task categories, M, in the experiments run; the y-axis shows the percentage of connecting trials, out of 100. Interestingly, for $M = 90$ and below trials are always connecting. From there the percentage of connecting trials drops off steeply so that very quickly in all trials agents never, or almost never, develop large cliques.

This behavior is advantageous from an engineering standpoint. It is almost certain that

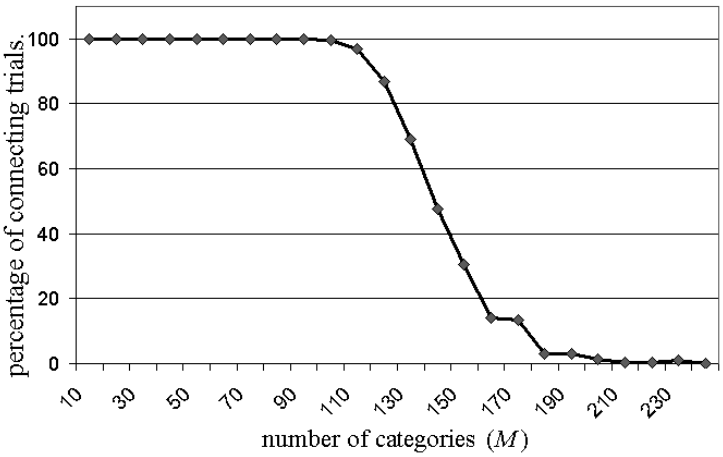


Figure 3.5: Percentage of trials that formed large cliques; $\delta = 3$, $N = 2000$.

systems with a larger M are probably not worth running, while in systems with a small M most agents will find matches for their tasks. We find later that this observation is independent of N, for large enough N. Ninety categories is a surprisingly large number, probably high enough for simple applications. We also show later that as the number of task matches each agent is searching for, δ, increases the number of categories supported by the system also increases. With six ports per agent the system supports up to 2000 categories.

It is also interesting to know the effect that the number of categories has on running time. Figure 3.6 shows that as the number of categories increases, the time it takes for connections to be found appears to increase linearly. Here the x-axis shows the number of categories, and the y-axis shows the number of turns run until no more changes occur in the system. We plot data from 100 trials at each point, showing the maximum, minimum and average values. The upper set of lines represents the trials that formed a large clique; the lower set of lines represents the trials in which agents remain unconnected.

In Figure 3.7 we compare some typical connecting trials with 10, 50, 90, 130 and 170 categories. This shows how the shape of the curve in Figure 3.4 changes as the number of categories, M, changes. The A region of Figure 3.4 in which initial small cliques are found, grows longer and the steep part of the curve in region B, during which the small cliques join together, becomes more gradual. The maximum percentage of connecting links also decreases, however we found in further experiments that this effect is lessened for trials with more agents. Trials with 120 categories and 2000 agents form between 92.5% and 94.9% of possible connections, for 20,000 agents this increases to between 97.7% and 98.2%.

These results on the effects of changing M are dependant on the fact that we considered a task distribution in which categories were picked at random with equal probability of each category occurring. If some categories are less likely to occur, however, others will become more likely. As it appears that the system is most affected by the ability of sufficient agents to find initial matches, changing the task distribution should not significantly effect

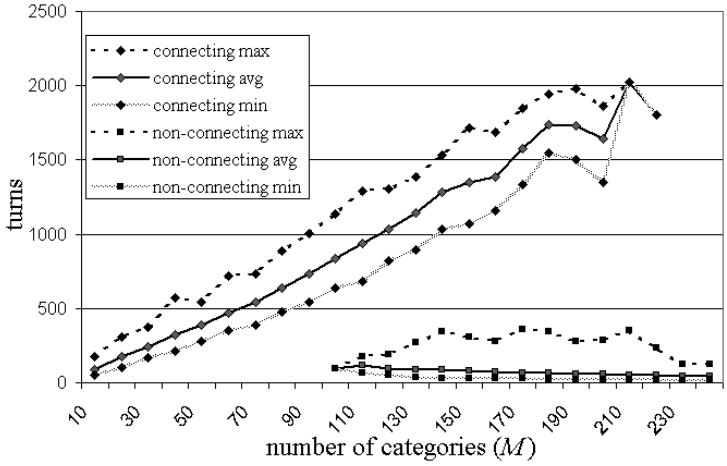


Figure 3.6: Time until connections stop forming; $\delta = 3$, $N = 2000$.

the overall behavior. In Section 3.5 we will consider this issue further by exploring more complex matching behaviors.

3.3.3 Trial Lengths

Considering the 300 category case in Figure 3.3, it is conceivable that it will eventually behave like Figure 3.4, the 80 category case, if only the system was run for longer. This could be especially true when the number of categories is just large enough that the system does not always connect. Perhaps considering longer trials would result in a higher percentage of connecting trials when running experiments with large numbers of categories.

In Figure 3.5 trials were terminated after 100 consecutive turns occurred in which no new connections were formed. We found that increasing this time to 400 or 800 turns produces no difference in the results. This is because when cliques are small the search space of their unmatched ports is also small. During 100 turns it is thus likely that all the possible combinations of link pairings have been tried. The search space for a clique only changes if it forms a new connection, or one of the cliques near it forms a new connection. If no cliques change during 100 turns none of the search spaces change. This means we can be fairly sure that the system has stagnated. This is a valuable property; it means that an individual clique has a guideline to determine if it is in a system, or a part of a system, that has come to a standstill. Agents could possibly use this fact to locally determine if they should switch to a more complex behavior to allow matches to be found.

3.3.4 Number of Agents

We next examine how well the system scales as the number of agents increases. Figure 3.8 shows the number of categories at which 50 percent of trials are connecting, for experiments with differing numbers of agents. Looking back to Figure 3.5, this occurs at just under 140 categories for $N = 2000$. We use the 50 percent point rather than the point at the end of the 100 percent success rate because it is easier to determine. The x-axis in Figure 3.8 shows the

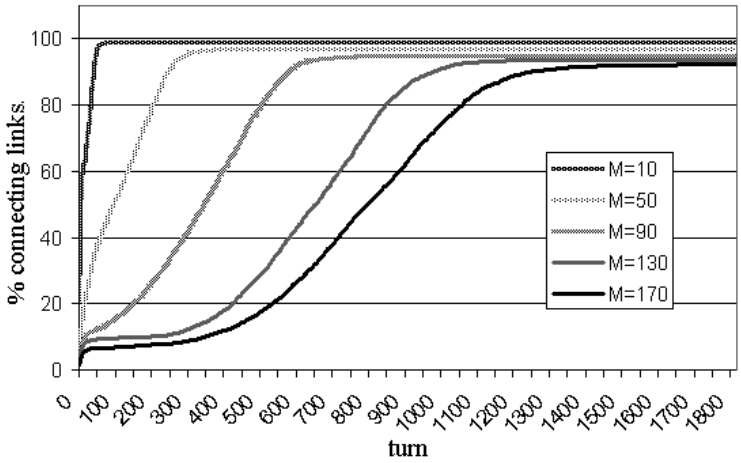


Figure 3.7: Percentage connecting links over time for varying M; $\delta = 3$, $N = 2000$.

$\log_2$ of the number of agents in the system; the y-axis is the number of categories at which 50 percent of trials, out of 100 run, form large cliques. From the graph it appears that the number of categories below which trials will be connecting increases slowly, approximately logarithmically with the number of agents in the system. This is likely to be due to the increased chance of an initial large clique forming somewhere, the larger the system is.

In Figure 3.9 the number of categories is fixed at 120 and trial length is plotted, that is the time until connections stop forming, for the connecting trials as a function of the number of agents. We find that this time increases slowly. If we could plot data points for larger numbers of agents we might even find a logarithmic relation, though we are unable to determine this relationship conclusively based on the given evidence. Based on Figures 3.8 and 3.9, we conclude that the system scales well with the number of agents.

3.3.5 Number of Ports

The final variable that can affect the basic matchmaking behavior of our agents is the number of ports per agent, δ. δ determines how large the search space of each agent is, especially at the start of a trial. Thus, it should have an effect on how many categories the system can support, that is the maximum value of M for which all trials are connecting. Figure 3.10 shows that the number of categories supported by the system increases rapidly as δ increases. Further experimentation also indicates that the general shape of the curves describing the speed at which connections are found, and the percentage of trials that connect for a particular value of M, remains the same as for $\delta = 3$, though they become less precise as the number of ports increases.

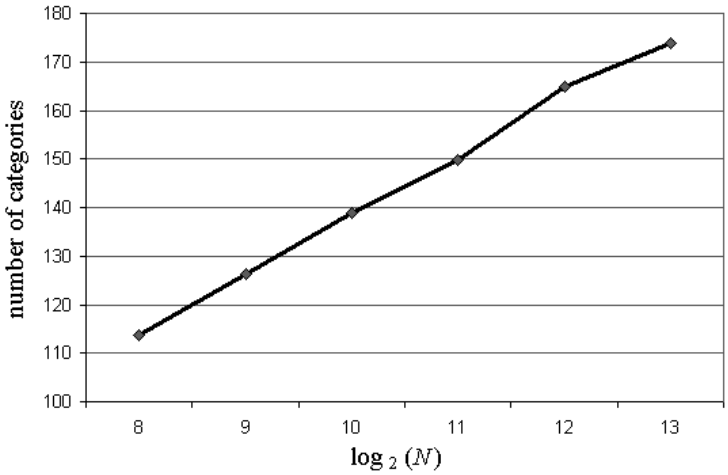


Figure 3.8: Number of categories (M) for which 50% of trials are connecting vs. N; $\delta = 3$.

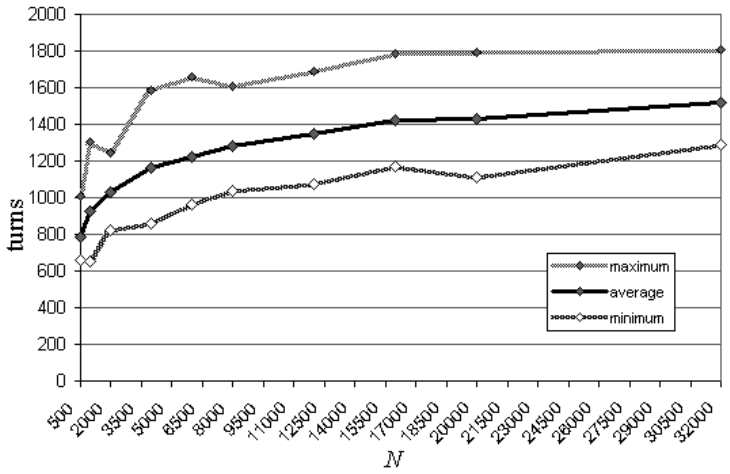


Figure 3.9: Trial length vs. number of agents; $\delta = 3$, $M = 120$.

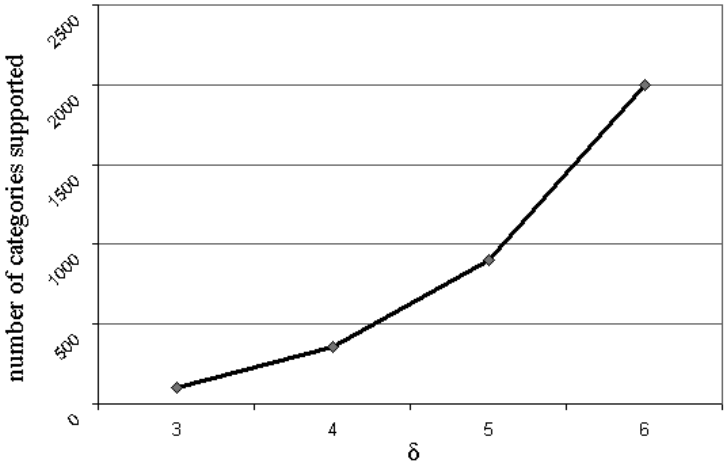


Figure 3.10: Number of categories supported vs. number of ports per agent.

3.4 Limiting Clique Size

Up to this point no limits have been placed on clique membership. Any agent which wishes to join a clique has been allowed to join. Allowing cliques to grow to any size presents a problem if swapping links within cliques is controlled centrally, since the cost of controlling the clique borne by the coordinator agent is proportional to clique size. For instance, as the number of agents in the clique increases the coordinator agent must deal with proportionally more swap requests. Swapping can be done in a decentralized manner. Simply decentralizing swapping, however, can only decrease the rate at which a clique finds possible matches with its neighbors since the expected time before a particular link between a given internal and external port is tested is at best proportional to the clique size. An alternative approach to maintaining decentralization is to limit the size of the cliques. In theory this could be done by downgrading connecting links, changing their status back to nonmatching, and thus potentially splitting a clique. The disconnect rate, that is the rate at which connections are downgraded, could then simply be adjusted to match the rate at which they form. In practice, however, this balance is hard to achieve. Setting a high disconnect rate stops most cliques from forming whereas lowering the disconnect rate, even slightly, still allows for large cliques.

An alternative, more direct, solution is to simply place a fixed limit on the number of agents in a clique, to reflect a limited amount of resources available to run clique operations. We thus place a limit on the maximum number of agents a clique may contain, S, and enforce this by means of the clique *requestConnection* procedure, as shown in the code provided in Equation 3.1 below. For now we will use a single *size limit* for all cliques, though naturally this could also be a parameter of individual cliques.

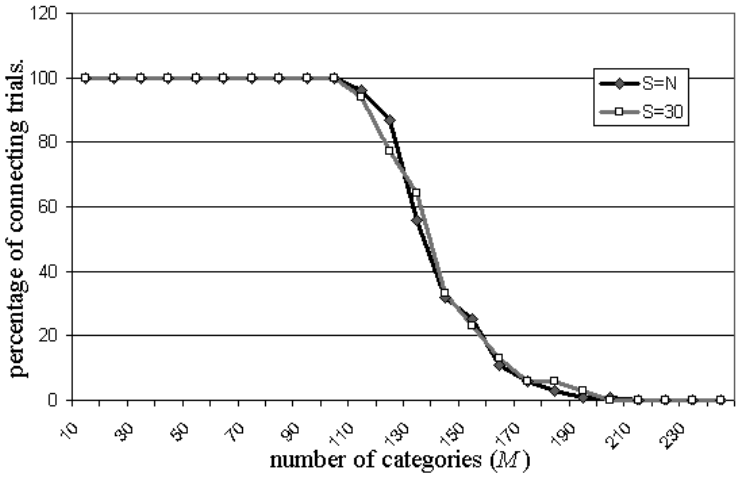


Figure 3.11: Percentage of trials, out of 100, that developed large cliques; $\delta = 3$, $N = 2000$.

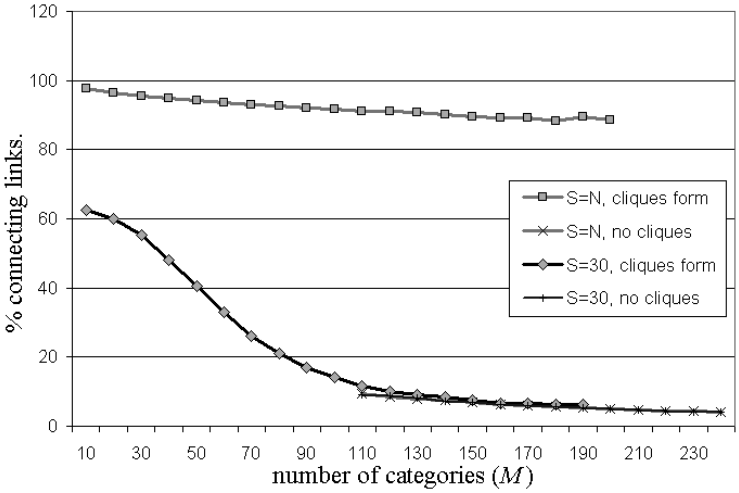


Figure 3.12: Average percentage of connecting links at trail end for trials in Figure 3.11.

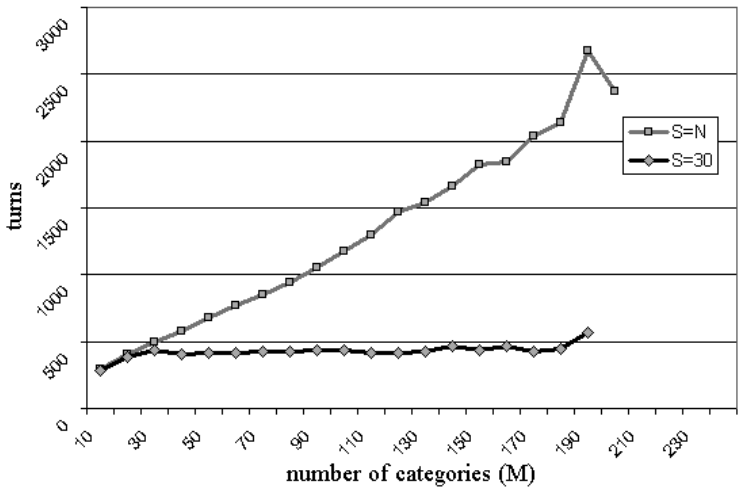


Figure 3.13: Average time until connections stop forming; $\delta = 3$, $N = 2000$, 100 trials per point.

```
[Clique] external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when
(``available'')
  if (members.size + requester.members.size ≤ S ) return ACCEPT;
  else return REFUSE;
}
```
(3.1)

Figures 3.11, 3.12 and 3.13 shows how the basic matchmaking behavior described in Section 3.3 changes when clique size is limited to 30 agents. Limiting cliques to between 10 and 100 agents provides similar results. The proportion of connecting links for 100-agent cliques is between 1 and 7 percentage points higher than those shown for cliques of size 30. When clique size is limited the difference between connecting trials and those that do not form large cliques is less clear-cut. We separated trials into those that formed a maximum-sized clique and those that did not. Figure 3.11 shows that the size-limited system supports the same number of categories. On the other hand, Figure 3.12 shows that as the number of categories increases the percentage of connecting links in a trial that forms a "large" clique decreases towards that of trials that do not form large cliques. We find, however, in Section 3.6 that this limitation does not make as much of a difference in a dynamic system, in which tasks are replaced. Figure 3.13 shows that though size-limited cliques find fewer connecting links, they also take less time to reach their final configuration.

3.5 Flexible Matching

In the experiments above we considered task categories from a set $T = \{t_1, \ldots, t_M\}$, represented as integers, $[1 \ldots M]$, in our pseudo-code. Matches between these tasks were made deterministically. We found that the number of these *exact categories* that the system could contain was limited, depending on the number of ports per agent. In more realistic situations, however, the matching of tasks is likely to be more flexible. It is thus important to

know what effect matching functions that accept a range of matches for a task, possibly with varying probabilities, will have on the range of tasks supported by the system.

To study this, in place of using integer categories which each match to one other category, we choose categories from a continuous interval and allow matches to be made to a range of other nearby categories. We choose categories from an interval $[0,M)$, and use a matching function that defines a link's strength (strength is equivalent to similarity since all agents use the same function) based on the distance between the two task categories of its adjacent ports, t_i and t_j. We call this distance the *length* of the link. We use a simple cyclic distance: $d(t_i,t_j) = \min\{|t_i - t_j|, M - |t_i - t_j|\}$. This cyclic distance is used to avoid edge effects, and gives a maximum distance of $M/2$. Link strengths are defined as $m(t_i,t_j) = M/2 - d(t_i,t_j)$ so that identical categories form the strongest links.

```
object Task{
   type: Real; //Chosen out of [0, M])
}
```
(3.2)

```
[Port] internal procedure distanceTo(t2:Task) returns (Real){
  return ``some measure of distance between this port's
  task.type and t2.type'';
}
```
(3.3)

Furthermore, we consider σ_{low}, the threshold strength above which a link will be made into a match, to be chosen each time a new link is formed according to some probabilistic function. σ_{low} thus becomes a random variable from $[0,M/2]$. The distribution of σ_{low} determines how likely it is that an agent looking for a service of category t will accept a candidate service of category t'. The probability that a link between tasks of categories t_i and t_j will become a match is thus the probability that σ_{low} will be less than or equal to $d(t_i,t_j)$. We can thus define a *match probability* function $p_{match}(d)$ which gives the probability of a link of a given distance becoming a match.

```
[Port] internal procedure portMatches( p2:Port ) returns ( Boolean ){
  return (if ( ``true with probability p_match(distanceTo(p2.task))'' ));
}
```
(3.4)

```
[Agent] internal procedure tasksMatch( p1:Port, p2:Port ) returns ( Boolean ) {
  return p1.portMatches(p2);
}
```
(3.5)

In the following sections we look at several different match probability functions, first to compare against the exact case, and then to determine the effect of changing the shape of the probability distribution. In this way we model the fuzziness with which tasks may be matched in an application. We consider functions, p_{match}, that are non-increasing and have a maximum at 0, representing the notion that agents are more likely to accept offered services the more similar they are to the one desired.

Original Agents vs. Rectangular p_{match}

We first compare the original exact matching function to its closest match probability function for continuous task types, the uniform or rectangular distribution.

$$p_{match-rectangular}(d) = \begin{cases} 1, & \text{if } d < d_{max} \\ 0, & \text{otherwise.} \end{cases}$$

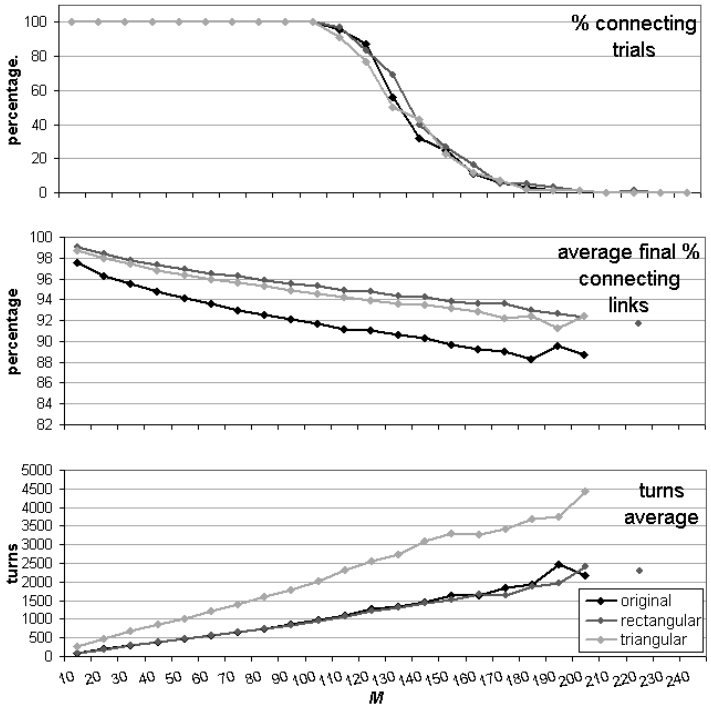


Figure 3.14: Original exact matching compared to $p_{match-rectangular}$ and $p_{match-triangular}$; $\delta = 3$, $N = 2000$, $S = N$, $d_{max} = 1/2$, $p_0 = 1$, averages over 100 trials per point.

With this function two tasks within a certain distance of each other, d_{max}, always match. We set $d_{max} = 0.5$, giving each category a matching interval of size 1, equivalent to the interval of the matching function for integer tasks we studied earlier.

Figure 3.14 compares data for the original agents with agents using $p_{match-rectangular}$. As in the original system, $p_{match-rectangular}$ represents a case where a category will always match to any category within a certain interval of length 1. We see that the main difference is that the $p_{match-rectangular}$ case improves on the percentage of connecting links at the end of a trial by 1.4% to 5.3%. This is due to the fact that if a task A has two matches, B and C, B and C are unlikely to be identical and thus while some of their matching space overlaps there are also further tasks that will match only B or match only C. Thus B connecting to A removes less of C's potential matches than it does in the exact case.

Triangular p_{match}

In a more realistic system it is likely that an agent is willing to accept a range of related services as modelled in by $p_{match-rectangular}$ above, but also that the agent will prefer some of these services to others. This can be modelled by means of a triangular shaped matching function, with a high probability of matching, p_0, at distance 0 which decreases linearly to some maximum distance, d_{max}, at which a match could possibly be made.

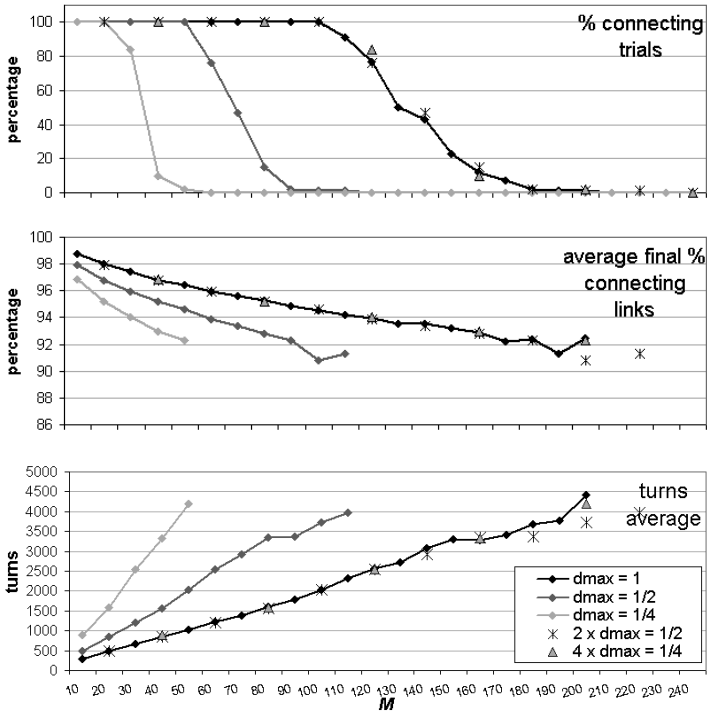


Figure 3.15: $p_{match-triangular}$ with varying d_{max}; $p_0 = 1$, $\delta = 3$, $N = 2000$, $S = N$, averages over 100 trials per point.

$$p_{match-triangular}(d) = \begin{cases} p_0(1 - d/d_{max}), & \text{if } d < d_{max} \\ 0, & \text{otherwise.} \end{cases}$$

This triangular function provide a simple representation of an agent's preference for a particular category, with a diminishing ability to accept related categories depending on how similar they are to the desired category. For comparison to $p_{match-rectangular}$, Figure 3.14 includes data for agents using $p_{match-triangular}$ with $p_0 = 1$ and $d_{max} = 1/2$, labelled 'triangular'. Agents using $p_{match-triangular}$ in general perform less well than the agents using $p_{match-rectangular}$. $p_{match-triangular}$ supports the same number of categories, but with a more gradual drop off and final cliques have 0.1% to 1.5% fewer connecting links. More importantly however, trials are 1.8 to 2.9 times longer due to the fact that task pairs that may possibly match might meet a number of times before doing so. We further explore the effect of varying p_0 and d_{max}, first setting p_0 to 1 and varying d_{max} to determine the effect of the support of the matching function. We then fix d_{max} at $1/2$ and vary p_0 to determine the effect of changing the probability of two tasks matching.

Figure 3.15 shows the first of these experiments. All data sets show agents using $p_{match-triangular}$ with $p_0 = 1$. We ran experiments using $d_{max} = 1/2$, $1/4$, and $1/8$. The points labelled "$2 \times d_{max} = 1/2$" show the data for $d_{max} = 1/2$ with its category axis dou-

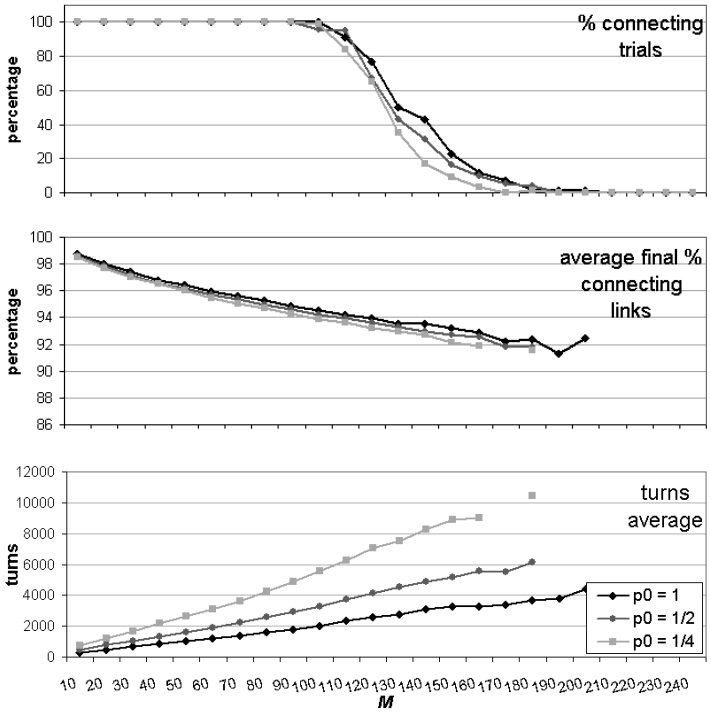


Figure 3.16: $p_{match-triangular}$ matching functions with varying p_0; $d_{max} = 1/2$, $\delta = 3$, $N = 2000$, $S = N$, averages over 100 trials per point.

bled, similarly "$4 \times d_{max} = 1/4$" shows the data for $d_{max} = 1/4$ with its category axis quadrupled. As these points indicate, multiplying the support of the match probability function by a constant simply has the effect of multiplying the supported category range by the same constant. This makes sense, categories are chosen from an interval so scaling the matching function's support is equivalent to scaling the interval. We next look at the effect of varying p_0. In Figure 3.16 $d_{max} = 1/2$ for all data sets and we plot trials for p_0 values of 1, $1/2$ and $1/4$. Reducing the probability of matches being made has a small effect in reducing the number of connecting trials, but most importantly changes the speed at which the system finds matches. Compared to trials with agents using $p_0 = 1$, trials with $p_0 = 1/2$ take 1.5 to 1.7 times longer and trials with $p_0 = 1/4$ takes 2.4 to 2.8 times longer to find matches.

"Fuzzy" p_{match}

In our final experiment with flexible matching we look at the effect of the "fuzziness" of the match probability function on our system. For the triangular match probability function matches of a length near to d_{max} are unlikely to be made. In the top graph in Figure 3.14, however, there appears to be little difference in the maximum value of M for which trials connect between a system using the triangular match probability function and one using the rectangular match probability function. This indicates that even matches that are unlikely to

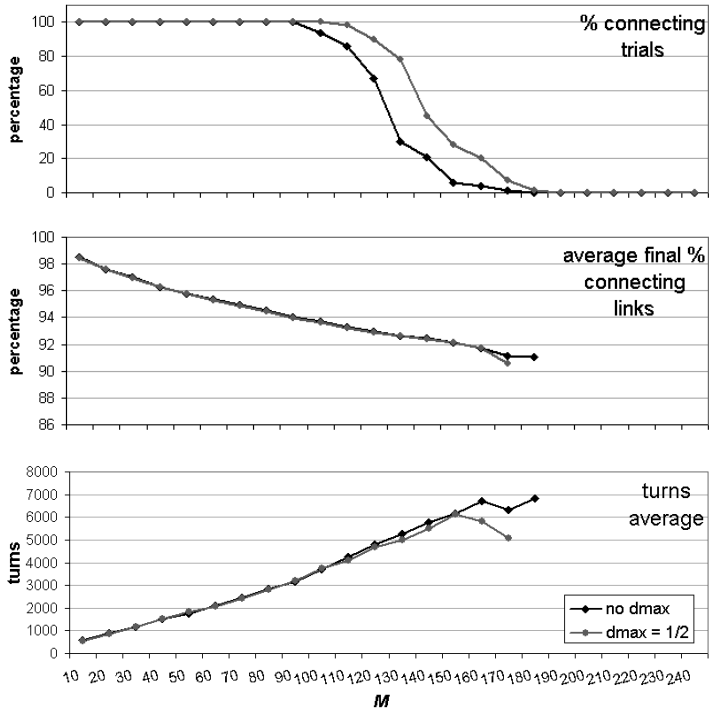


Figure 3.17: $p_{match-scaled-normal}$ with d_{max} set to ∞ and $1/2$; $\delta = 3$, $N = 2000$, $S = N$, averages over 100 trials per point.

be made still play a role in determining how large M can be. To investigate this further we consider a case where agents have a markedly preferred match for a task, but could possibly choose matches that are very different from their preferred partner. We use the the normal distribution function to model this behavior. Experimentally we determined that for this function a mean of 0 and a variance of $(11/2)^2$ gives approximately the same maximum value of M for which trials connect as our previous experiments.

$$p_{match-scaled-normal}(d) = \begin{cases} \frac{1}{\sqrt{4\pi/11}} e^{-(d/11)^2}, & \text{if } d < d_{max} \\ 0, & \text{otherwise.} \end{cases}$$

Figure 3.17 shows data for agents using $p_{match-scaled-normal}$ with d_{max} set to infinity compared to the same agents with $d_{max} = 1/2$. At distance $1/2$ the probability of two tasks matching is already very low so the tail of the distribution which is cut off by setting $d_{max} = 1/2$ accounts for only 1% of the total area under the original curve. We see, however, in the top graph of the figure that cutting off this tail still affects the the maximum value of M for which trials connect. The value of M for which 50% of trials connect is about 12% lower when d_{max} is limited to $1/2$. On the other hand, limiting d_{max} in this way makes little difference to the length of trials or the final number of connections found in connecting

trials.

3.6 Task Replacement

Up to this point we have only modelled systems with a fixed initial set of tasks to be matched. In more realistic matchmaking settings additional queries will be made to replace those for which an answer has been found. In this section we study the behavior of the model when given a dynamic set of tasks to determine if matches for new tasks can be found once initial cliques have been formed.

We add to the system's parameters a probability, P_e, of matched tasks, joined by a matching or connecting link, ending at a given turn. This represents jobs that take some random amount of time to complete. When a task between two agents completes, each agent is given a new task of a randomly selected category. The link between these two new tasks is left in place, but downgraded to nonmatching. Breaking a connection in this way can result in a clique splitting into two smaller cliques.

```
[Clique] process reconsiderCompositon() [for each m in members]{
  whenever (``clock phase 4'')){
    for each aq in m.acquaintances {
      if ( ``true with probability Pe'' ){
        aq.reset();
        ``split off cliques if necessary'';
      }
    }
  }
}
```
(3.6)

```
[Port] external procedure reset(){
  connecting := FALSE;
  matching := FALSE;
  task := ``select new Task'';
  neighbor.reset();
}
```
(3.7)

Figure 3.18 shows sample trials with 2000 agents, 60 categories, a clique size limit of 30 and $P_e = 0.003$, for the original agents with exact task categories and agents using the $p_{match-rectangular}$ match probability function. For the exact case, things go well for a time, but the number of connections in the system gradually decays and eventually the cliques fall apart. Figure 3.19, which shows the category distribution of tasks at the start, middle and end of the trail, shows why. Task category pairs are setup so t_0 matches t_1, t_2 matches t_3 etc. At the start of the trial the distribution of task categories is approximately even. At the end, however, there are disproportionately large numbers of some category types. Since the number of possible matches that can be made between tasks of two matching categories is equal to the minimum number of tasks of either category, the number of possible matches in the system is greatly reduced. The unevenness in the number of tasks of each category occurs because matches are made, removed from the system, and replaced with new random tasks. For any pair of matching categories, a random distribution with an increasing number of members will eventually contain a large absolute difference in the number of members of each category. The agent matchmaking system needs to include a minimum number of possible matches, as indicated by the maximum value of M for which trials connect. This means that cliques will always eventually fall apart if tasks are replaced in this manner.

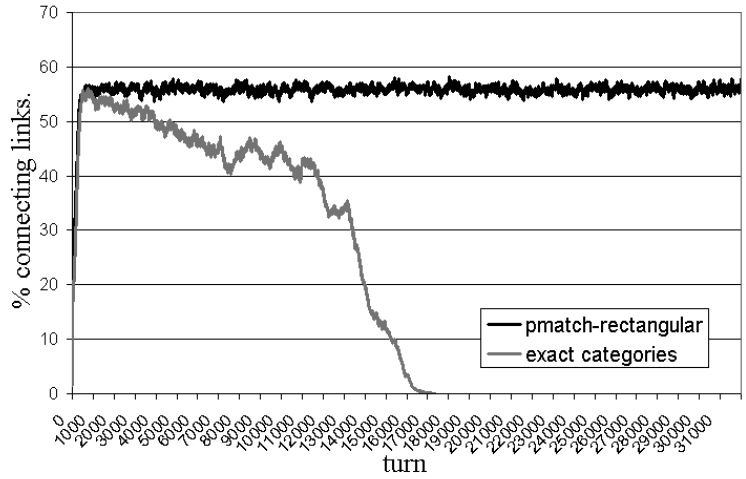


Figure 3.18: Percentage of connecting links over time; $P_e = 0.003$, $\delta = 3$, $M = 60$, $S = 30$, $N = 2000$.

The system with the $p_{match-rectangular}$ match probability function avoids this problem since a category is able to match itself. This system can thus continue to form connections indefinitely. The same flexibility can be achieved for the exact category case if agents drop tasks for which they cannot find matches with a certain probability, in exchange for new ones. Accordingly, we define a new parameter P_c, the probability that an unmatched task will change to a new type at each turn. Figure 3.20, which shows sample trials with various P_c values indicates that with this mechanism cliques no longer fall apart. Figure 3.21 shows a longer trial for P_c = 0.00005. This trial is interesting since it shows a system where cliques form, change, and occasionally completely break apart then form again. Note that in the static case setting $S = 30$ produced trials where only about 33% of links became connecting, whereas in the dynamic case the system reaches a level of 55% connecting links. In a dynamic setting the number of connections at any one time however is not as important. Looking at the tasks completed per agent over time we find that all agents complete a comparable numbers of tasks.

3.7 Conclusions

From the observations made in this chapter we conclude that our agent procedure can be used to do matchmaking in a decentralized manner, under the right conditions, and that this form of matchmaking could have some advantages in terms of scalability over centralized matchmaking. We draw the following preliminary conclusions which we will use to guide the work in later chapters.

- The majority of agents find matches, as long as the range of task types, M, is limited.
- Matches are found quickly. This behavior appears to be fairly stable, changing the model in various ways distorts the rate at which matches are found, but causes no

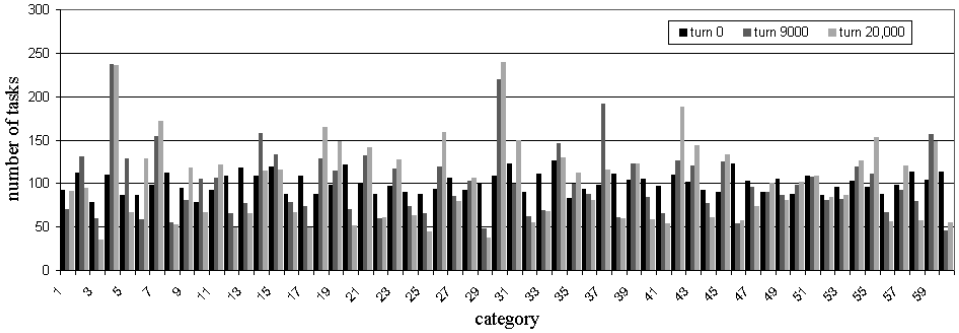


Figure 3.19: Category distribution for the exact category case in Figure 3.18.

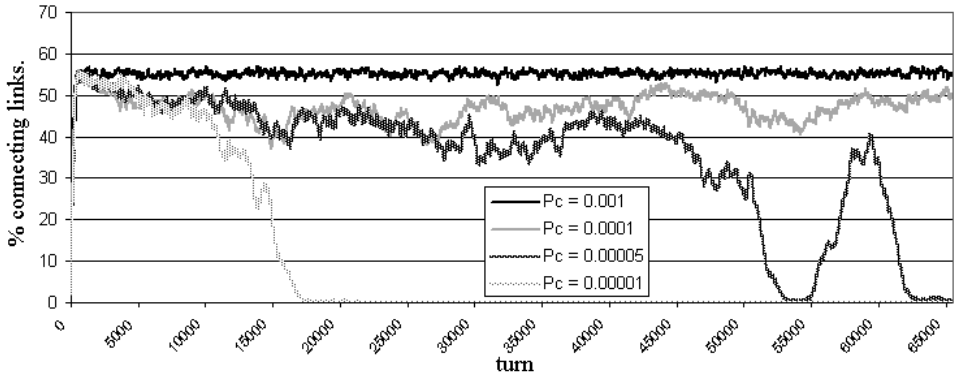


Figure 3.20: Percentage of connecting links over time for various values of P_c; $P_e = 0.003$, $\delta = 3$, $M = 60$, $S = 30$, $N = 2000$.

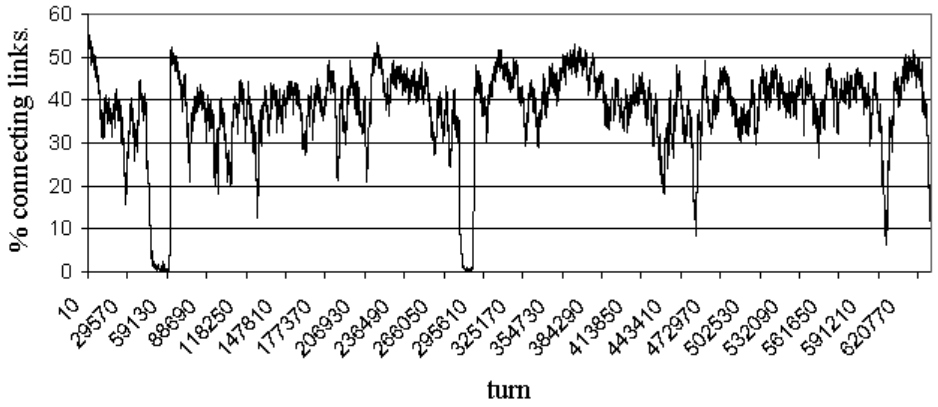


Figure 3.21: Percentage of connecting links over time; P_c=0.00005, $P_e = 0.003$, $\delta = 3$, $M = 60$, $S = 30$, $N = 2000$.

fundamental changes to the way the system behaves. Changes we tried include increasing the size of agents' neighborhoods, decreasing the size of cliques, changing the nature of the function by which matches were determined, and varying the time at which new tasks were introduced.

- The system appears to be able to handle large numbers of agents and scales well, provided that the cost of clique operations is kept limited.
- There are two critical factors that limit the applicability of the model, the ratio of the matching range to the range of task types in the system as a whole (M) and the size of cliques.
- The ratio of task ranges is a measure of the probability that an agent's task matches a task in one of its neighbors. We can manipulate this in two ways, by increasing the flexibility with which matches are accepted and by increasing the number of links/neighbors per agent.
- The system has a strong tendency to form large cliques. To keep cliques manageable their size is best controlled by directly preventing larger cliques from merging. This indicates that the model is most suitable for applications in which clique size can be kept relatively small.

Chapter 4

Auctions

In the previous chapter we found that, in the abstract case, peer-to-peer agents could find matches for tasks, provided that the chance of two random agents matching was high enough and cliques could be limited in size. In this chapter we demonstrate for a particular application, an agent auction, that these limitations are not overly restrictive. The work in this chapter was originally presented in [33].

4.1 Introduction and Related Work

Auctions are a widely studied form of matchmaking through which self-interested traders are able to settle on a fair price for a commodity. These traders are quite naturally representable as agents, and thus agent auctions are used to explore many different fields. These fields include Economics where the efficiency of different markets is measured, Game Theory where the abilities of market mechanisms to stand up to malicious agents are compared, and finally Computer Science where agents are used to buy or sell goods under varying circumstances [23] [38]. Additionally, the simplicity and robustness of agent auction algorithms make them well suited to a variety of applications: e-commerce naturally, but also more general resource allocation problems [50] [51]. From the point of view of a study of matchmaking, auctions have the advantage that the individual matching, or *bidding*, functions needed by participants can be fairly simple and are well understood. Moreover, auctions illustrate the general principle that individual autonomous behaviors can lead to a useful overall group behavior. This principle is often used to motivate studies of decentralized systems.

On the other hand, while an auction demonstrates distributed determination of prices, auctions are most often implemented using a central auctioneer and therefore are not in fact fully decentralized systems. This central auctioneer distributes global information about current prices and deals made among traders, which is subsequently used in individual traders' bidding functions. In a system running on a single machine or a well-connected network, such high-quality information is certainly worth the cost of maintaining a central source. As systems move to run on less reliable networks, however, the communication costs of maintaining a central auctioneer could become prohibitive, limiting the number of auction participants.

In this chapter we adapt the agent matchmaking procedure in Chapter 3 to implement a peer-to-peer agent auction. Auctions are an ideal application area in which to test our

matchmaking agents, for several reasons. First, as agent cliques have no actual purpose within an auction, the limit, S, we place on clique size to keep distribution manageable is not a drawback. Second, basic auctions deal with commodity goods, meaning that price is the only factor in determining matches. Agents can be highly flexible on price, and thus the limitations on the task range, M, are also of no great concern. Finally, there are a number of existing bidding algorithms that we can adopt as the abstract matching functions required by the agent matchmaking procedure.

Agent auction research is based on the core theory that markets produce an efficient allocation of resources even when using agent traders in place of human traders. Economic theory states that there is a computable equilibrium price for a given commodity market based on the best price at which each trader is willing to buy or sell. This equilibrium price is the price at which the largest number of trades will be made. Remarkably, human markets settle on this equilibrium price after a time, in spite of the fact that none of the traders in the market know the other traders' best prices. One line of research, which is particularly interesting to matchmaking, explores the question: what is the minimal intelligence required in market participants to create an efficient market? Gode and Sunder first asked this question in [12] where they compared "zero intelligence" agents that bid randomly between lower and upper bounds, to experiments with human traders done by Smith [43]. They concluded that the market mechanism itself, and not the intelligence of the traders, was sufficient for the market to settle at equilibrium price. Cliff and Bruten [2] later investigated this question further and showed that such zero intelligence agents work only given certain supply and demand curves. They created "zero intelligence plus" agents that use a simple learning algorithm to make use of past information, and showed that it worked in the circumstances where Gode and Sunder's zero intelligence traders failed. Priest and Van Tol [37] later extended this work from an auction mechanism where only one agent bids at a time to a more realistic scenario where all agents bid simultaneously.

The only concern here is that the bidding behavior of "zero-intelligence" agents, like most agent bidding algorithms, includes a notion of a central auctioneer. It is therefore assumed that agents know some global statistics about the current bids and offers, specifically the best bids and offers in an auction. The cost of maintaining this information in all agents is not inconsiderable, and could be prohibitive for applications involving large numbers of agents distributed over a network. It requires that one agent, the auctioneer, collects all bids and regularly distributes summaries to all participants. In the decentralized setting we therefore extend the question about the minimum agent abilities for an efficient auction to include the issue of the *minimum communication requirements* among agents.

4.2 Model

Markets consist of a number of *traders*; *sellers* who have an item that they wish to sell, and *buyers* who have some money to buy items. In this chapter we consider a simplified theoretical market with only one commodity good being traded. This means that all items for sale are identical, avoiding the question of one item being intrinsically more valuable than another. Traders are modelled as having a *reservation price*, a minimum price at which they are willing to sell items, or a maximum price they are willing to pay for items. The reservation prices of all traders in the market put together create supply and demand curves,

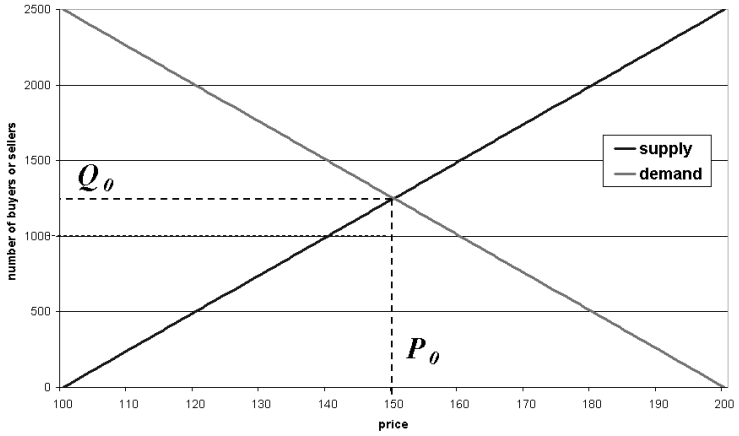


Figure 4.1: Example supply and demand graph.

as shown in Figure 4.1. The supply curve shows the number of items sellers are willing to sell at each price, the demand curve analogously shows how many items buyers are willing to buy at each price. The intersection of the two curves is at the *equilibrium price*, P_0, and *equilibrium quantity*, Q_0. This *equilibrium point* is the price and quantity at which the maximum number of items will be exchanged. In theory, markets naturally tend towards trading at this point. If the market price is above equilibrium more sellers will compete to trade with fewer buyers thus bringing the price down. Vice versa, if the price is below equilibrium there will be more buyers wanting to buy than items for sale, driving the price up. Therefore, a measure of the effectiveness of a market mechanism is how close to equilibrium trades take place. Also, since markets with no prior history will most likely start making trades off equilibrium, a second important measure is how quickly equilibrium is reached. In this chapter we consider agents trading in a *continuous double auction*. In auctions market prices are determined through *bidding rounds*. During a bidding round all buyers and sellers shout the current price at which they are willing to exchange an item. If these prices overlap the traders involved make a *deal*, otherwise the traders need to update their price to shout in the next round. In continuous double auctions both buyers and sellers announce their current asking prices, and shouts remain valid until updated by a new shout in a later round.

In the following experiments software agents act as traders in simulated auctions. Each trader agent represents a single buyer or seller with a fixed reservation price and a single item to buy or sell. Traders update their asking prices according to a simple learning algorithm. Once a trader has made a trade it re-enters the system, with a new item or money and the same reservation price, with probability P_r at each bidding round. We compare these trader agents trading in two different auction setups, a basic centralized auction and a peer-to-peer auction in which traders are used as the objectives in the matchmaking procedure from Chapter 3. Our aim is to determine how effective a peer-to-peer auction might be in reaching equilibrium, and to compare the communication costs of finding trades in the two

setups. The peer-to-peer auction should have an advantage in terms of communication because is spreads communication equally among all nodes, rather than disproportionately to the central auctioneer. We are thus interested in measuring not the total volume of communications that take place during an auction, but rather the estimated time required considering that communication between separate pairs of agents can take place in parallel.

We estimate communication cost by running simulations and recording the number of bidding rounds that traders take to discover a market price near the theoretical equilibrium price. We calculate or measure the number of messages, or direct communications between two agents, or between an agent and an auctioneer, that would have been required to implement these bidding rounds in a distributed system. In this setting we consider each agent and the auctioneer to be processes run on separate machines. Messages between separate pairs of machines can be sent and received simultaneously during a bidding round. Therefore, instead of measuring the total number of messages that must be sent, we measure the number of *message rounds* or sets of simultaneous messages that must be sent to accomplish each bidding round. The number of message rounds needed to reach equilibrium is taken as an estimate of the sequential time required by each auction type.

It is, however, difficult to simulate all of the complications of a truly asynchronous network. For this reason we make some simplifying assumptions to keep our experiments manageable. We assume messages always arrive in zero time, avoiding the issue of lost or delayed messages and allowing us to assume that if a message has been sent it has been acted upon. Since we are most interested in the overall mechanisms of the two auction types, and removing these assumptions will effect both systems, these assumptions should not have a great impact on the general comparison being made. The effect of latencies and lost messages on a node in a real distributed environment should be roughly proportional to the number of messages the node receives, thus these measures do not change the division of work between nodes. Long latencies could have a somewhat greater effect on the centralized system since all nodes must be delayed while the auctioneer waits for bids, while in the peer-to-peer system only the current partner node is delayed by a message being delayed. If we assume lost messages are ignored, their effect will be to simply lower the quality of market information. This might slow down market convergence, and thus could have a greater impact in the peer-to-peer system where information quality is already low.

Additionally, for the peer-to-peer auction we use the centralized version of time in which cliques perform the sequence described in 2.5.1 once each turn, simultaneously. This gives us an easy time period for taking measurements and making comparisons to the central auction. It, however, avoids issues of cliques acting at different speeds; instead we simulate all cliques acting at the speed of the slowest clique. Thus we do not consider that smaller cliques could actually handle more bids in this time period, or that some cliques could have to wait for replies from slower ones. Overall, however, this restriction simply has the effect of slowing down some cliques. If we make the assumption that a random sample of the traders are effected in this way, the remaining traders in the market will have similar supply and demand curves to the total set of traders. Taking a random sample of traders lowers the number of trades possible, and thus changes Q_0, but it does not change the distribution of prices, and thus P_0 remains the same. Therefore, slowing the actions of some cliques should only decrease the rate at which market equilibrium is reached, not change the overall behavior of the market.

4.2.1 Central Auction & Bidding Algorithm

We use a modified version of the centralized auction and the agent bidding update algorithm presented in [37] as a basis for comparison. In that work an auction is divided into bidding rounds in which each trader who has resources available sends its current asking price to a central auctioneer. The auctioneer determines if there are overlapping bids, and if so pairs up trading buyers and sellers. The buyer with the highest bid is paired to the seller with the lowest offer, second highest bid to second lowest offer, and so on until there is no more overlap. Trades are made at the average of the asking prices of each paired buyer and seller. The auctioneer then broadcasts the best bid and offer to all the agents in the market. This information is used by the traders to determine what price to shout in the next round. Agents who have traded have their good or money replaced with constant probability P_r each round after having made a trade, allowing them to reenter the market.

If each trader agent and the auctioneer are considered to be separate entities, each able to send or receive one message at a time, the serial communication costs of this centralized procedure are high. During a bidding round traders first send bids to the auctioneer. These must be received and processed, giving a cost of up to N message rounds, where N is the number of traders. The auctioneer must then send a message to each trader in turn with the current market information and that trader's deal status. As traders continue to update their prices, even when they have no resources, this cost is always N messages. There is thus a total cost of up to $2N$ message rounds per bidding round. This cost can be reduced to $O(\log N)$ by using a distributed auctioneer, however as this involves other complications we leave this analysis until Section 4.5 and for now consider the simpler single auctioneer.

The bidding update algorithm used by the trader agents in [37] is a heuristic update rule, with a simple learning element, designed to demonstrate that even very simple agents can be used to create effective markets. Trader agents are buyers or sellers, each with a single fixed parameter, their reservation price R_0, and a variable current asking price $p(t)$. This asking price is the amount shouted each bidding round. The asking price is initialized with a value, $p(0)$, which is a random value between the reservation price and a minimum for buyers, or a maximum value for sellers. Asking prices are updated each around according to the function:

$$p(t+1) = \begin{cases} \max\{p(t)+\Gamma(t+1), R_0\}, & \text{for sellers} \\ \min\{p(t)+\Gamma(t+1), R_0\}, & \text{for buyers,} \end{cases}$$

$$\text{where} \quad \Gamma(t+1) = \gamma\Gamma(t) + (1-\gamma)\beta(\tau(t)-p(t)), \text{ and}$$
$$\Gamma(0) = 0.$$

The function Γ is a learning rule used to update $p(t)$ based on a current target price, $\tau(t)$. It uses two variables, the learning rate, $0 \leq \beta \leq 1$ which determines to what extent the new price is based on the target price for this round, and the momentum, $0 \leq \gamma \leq 1$, which determines to what extent the current change in price should be the same as that in the last round.

Each round trader agents are given the minimum selling price shouted, s_{min}, and maximum buying price shouted, b_{max}. They use this information to determine the target price,

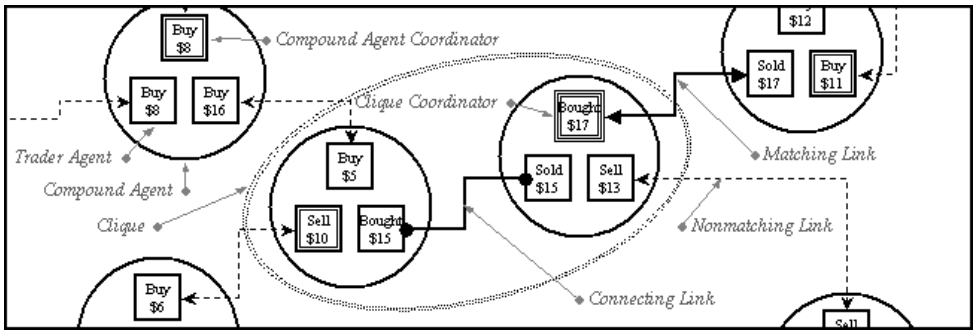


Figure 4.2: A snapshot of peer-to-peer trader agents.

$\tau(t)$, according to the following heuristic:

$$\tau(t) \quad = \quad \begin{cases} \begin{cases} b_{max}+\varepsilon, & \text{if } s_{min} > b_{max} \\ s_{min}-\varepsilon, & \text{if } s_{min} \leq b_{max} \end{cases} & \text{for buyers} \\ \begin{cases} s_{min}-\varepsilon, & \text{if } s_{min} > b_{max} \\ b_{max}+\varepsilon, & \text{if } s_{min} \leq b_{max} \end{cases} & \text{for sellers,} \end{cases}$$

where $\varepsilon = r_1 p(t) + r_2$ and r_1 and r_2 are uniformly distributed random variables taken independently from the intervals $(0, R_1]$ and $(0, R_2]$ respectively. The parameters R_1 and R_2 are used to define a small amount of random variation designed to model differences among traders.

In [37] the authors found that the above method of allowing agents to learn asking prices resulted in trader agents quickly finding market equilibrium for a variety of supply and demand curves, including the simple one which we study in this chapter.

4.2.2 Peer-to-Peer Auction

We create a peer-to-peer version of the central auction described in Section 4.2.1 by considering each trader agent to be an objective of a matchmaking agent from Chapter 3. We will thus consider the agents from our general model to be compound agents, each comprised of δ randomly chosen trader agents. Figure 4.2 diagrams the resulting system. Compound agents are represented by circles, each containing a number of trader agents. Trader agents are labelled as either buyers or sellers and pictured with their current asking price. Links join trader agents. A link is nonmatching if the adjacent agents are unable to trade and matching or connecting if a trade has been made. As before, connecting links join compound agents into cliques.

In previous chapters we have shown that by working together in a clique, resource-limited agents are able to gain a wider view of a system than is available in the small neighborhood they are able to maintain themselves. For peer-to-peer trader agents, working in cliques is thus advantageous since it increases the number of potential trading partners they have, and thus makes it more likely that they will find a fair price at which to trade. Up to

now, however, we have not discussed exactly how clique coordination is accomplished. Coordination between clique members is an important component of the communication cost of the peer-to-peer auction. In this chapter we implement this coordination by assigning for each clique a specific agent to be the *coordinator* of the clique. The clique coordinators maintain a map of the state of all traders in their clique in order to implement clique operations such as swapping links, adding members, and removing members. Similarly, since trader agents are considered to be the basic units of computation, for each compound agent one trader agent is assigned the role of coordinator for that compound agent.

Peer-to-Peer Auction Operations

Figure 4.3 provides pseudo-code for the peer-to-peer auction. The main object in this code is the Trader (lines 9-20) which replaces the Task object in the matchmaking code. The Agent object (lines 21-29) represents compound agents made up of δ Traders. Notice that Agent is therefore a virtual object in this pseudo-code version. Cliques (lines 30-60) remain basically unchanged, except that we add some mechanisms for coordinating intra-clique actions.

We use the same initial setup as before, creating random nonmatching links between traders. From the initial setup bidding rounds, or turns, proceed as follows. A trader first exchanges its current asking price with its neighbor (line 27). If the two are a buyer and a seller and their asking prices overlap a trade is made at the average of the two prices. They then update their asking price according to the update algorithm used in the centralized auction, described in Section 4.2.1. In a distributed setting, however, the best bid and offer in the system is no longer provided. Instead, agents simply use the last seen bid and offer, or their own price if it is better than one of these. (We also experimented with traders remembering the last m bids and offers seen, however we found that this merely slowed convergence.)

Two traders making a deal results in a matching link forming between their compound agents. In this way the generic matching function is accomplished by means of the bidding negotiation process. As before, compound agents group into cliques which internally rearrange nonmatching links. Each time a new link is created the associated traders exchange bids and offers, and update their asking prices as described above, mimicking the bidding rounds of the central auction.

To form cliques (lines 38 to 51), all matching links are upgraded to connections, with the restriction that cliques cannot grow beyond a maximum size, S (line 49). When a clique reaches this size trades continue to be made, but no new connections are formed. To limit the number of operations in which a clique is involved, we stipulate that each clique can only form one new connection per turn. This requires coordination among cliques to prevent a clique from receiving two or more simultaneous requests to form connections with others. We achieve this by having each clique designate one of its ports adjacent to a matching link, chosen at random each turn, as being able to form a connection (line 40). When two such neighboring ports are both designated by their respective cliques simultaneously the connection is agreed and they inform their cliques that they should merge.

Trader agents that have made a deal are considered to be out of the trading game for some time, as they no longer have an item to sell or money with which to purchase goods.

Thus, once formed, matching and connecting links persist for some amount of time. These links, however, are not permanent. Each bidding round, agents that have traded have their item or money replaced with probability P_r (line 55), which is analogous to p_{end} used in Section 3.6. For simplicity we assume that both the buyer and seller involved in a trade receive new resources at the same time. Since these traders are now ready to trade again the link between them is downgraded to nonmatching, possibly resulting in a clique being split (line 57).

Peer-to-Peer Auction Messages

The cost of a bidding round in the peer-to-peer auction is the total number of message rounds needed to implement it. To examine this cost we split each bidding round into four phases; shuffle, exchange, reply, and update. The messages needed in each of these phases depends on how virtual clique and virtual compound agent operations are accomplished. These phases thus vary slightly from the clock phases (1. Connecting, 2. Mixing, 3. Matching, and 4. Breaking) in the pseudo-code (as described in Section 3.2). Shuffle corresponds to the Clique *tradeIn* procedure returning new neighbors, exchange to the Trader *checkMatch* procedure, reply is used to accomplish Clique coordination and update is the Clique *searchForConnections* and *reconsiderComposition* operations.

In the following paragraphs we shall consider trader agents to be the basic units of computation, with one trader agent within each clique acting as a coordinator for clique actions. This coordinator is assigned in a fixed manner, all agents start out as the coordinator of a single agent clique, when cliques merge one of the previous coordinators becomes the new coordinator, and when a clique splits off a new clique the clique coordinator of the original clique appoints a coordinator for the new clique. When counting the message costs we assume that separate pairs of trader agents can send and receive messages in parallel during a message round, but that individual trader agents must process messages one at a time. We first describe each of the bidding round phases and calculate the number of message rounds that each phase must involve for a simple implementation, based on the pseudo-code. We then describe some optimizations that reduce the total number of message rounds required in a bidding round.

In the shuffle phase the clique coordinator gives each trader in its clique the information it will need during the bidding round. For all traders who have called *tradeIn* in the previous bidding round (line 36 of the pseudo-code) this information is an address for a new neighbor. In order to avoid a trader being given a new neighbor by its own clique at the same time as its address is given as a new neighbor to a trader in another clique we actually implement *searchForMatches* so that only one of the ports adjacent to a nonmatching link is traded in. This is done by having each link designate one port at random. In addition, during the shuffle phase one trader adjacent to a matching but nonconnecting link needs to be informed that it has been chosen to be able to form a connection (line 40 of the pseudo-code). In the worst case the shuffle phase requires the clique coordinator to sequentially send $(\delta S) - 1$ messages to each of the other traders in its clique.

In the exchange phase each unmatched trader sends its new neighbor its bid, and receives a reply offer. As the protocol for making deals, described in 4.2.1, is the same for all agents, each trader involved in this exchange can calculate independently if a deal has been made,

```
1.  object Port{
2.     trader:Trader; neighbor:Port; matching:Boolean; connecting:Boolean; myAgent:Agent;
3.    external procedure reset(){
4.      connecting := FALSE; matching := FALSE;
5.      ``give trader a new good or money'';
6.      neighbor.reset();
7.    }
8.  }
9.  object Trader{ //Tasks become trader agents
10.    type: [BUYER, SELLER];
11.    askingPrice: Real;
12.    external procedure  updatePrice( type:[BUYER, SELLER], price:Real ) {
13.      ``update askingPrice according to the bidding algorithm''
14.    }
15.    external procedure  checkMatch( t2:Trader ) returns ( Boolean ){
16.      updatePrice(t2.type, t2.askingPrice);
17.      t2.updatePrice(type, askingPrice);
18.      return ( ``true if this trader and t2 are an overlapping buyer and seller'');
19.    }
20.  }
21.  virtual object Agent{ //compound agents
22.    acquaintances: set of Port; //δ Traders per Agent
23.    c: Clique;
24.    process searchForMatches [for each aq in acquaintances]{
25.      whenever ( ``clock phase 3''  & aq.matching = FALSE & aq.connecting = FALSE ){
26.        aq.neighbor := c.tradeIn( aq.neighbor );
27.        aq.matching := aq.checkMatch( aq.neighbor.trader )
28.    }
29.  }
30.  virtual object Clique {
31.    members: set of Agent; //All agents that are members of this clique
32.    unwantedAcquaintances: set of Port;
33.    external procedure  tradeIn( p:Port ) returns ( p':Port ){
34.      unwantedAcquaintances.add( p );
35.      wait until( ``clock phase 2'' );
36.      return  unwantedAcquaintances.remove( ``random element'' );
37.    }
38.    process searchForConnections [for each m in members]{
39.      whenever  (``clock phase 1'' ){
40.        p:Port = ``choose any Port for which matching = TRUE and connection = FALSE'' ;
41.        if ( (p != null) & (p.neighbor.myAgent.c.requestConnection( self, p  ) = ACCEPT) ){
42.          p.connecting := TRUE;
43.          ``join the cliques'';
44.        }
45.      }
46.    }
47.   external procedure requestConnection(requester:Clique, port:P) returns (ACCEPT, REFUSE){
48.      if (``p.neighbor was chosen by searchForConnections in this clique''
49.        & (members.size + requester.members.size ≤ S ))return ACCEPT;
50.      else return REFUSE;
51.    }
52.   process reconsiderCompositon() [for each m in members]{ //New resource arrival
53.      whenever  (``clock phase 4'')){
54.        for each  aq in m.acquaintances {
55.          if ( aq.matching = TRUE & ``true with probability P_e''  ){
56.            aq.reset();
57.            ``split off cliques if necessary'';
58.          }
59.      }
60.    }
61.  }
```

Figure 4.3: Psuedo code for auction version of the agents.

and no further messages are needed. During the phase traders who have been designated as being able to form a connection can also exchange messages, requiring at most 2 sequential messages. Thus the total number of message rounds needed in the exchange phase is 2.

In the reply phase each trader must inform the clique coordinator of state changes made during the turn. Traders who have failed to make a trade need to send in the address of their current neighbor, in order to exchange it for a neighbor with which they might be able to make a deal. Traders that have traded, traders that have formed a connection, and traders that have received new resources must also all send a message to the clique coordinator. As the clique coordinator can only process messages one at a time, in the worst case the reply phase requires $(\delta S) - 1$ message rounds.

Finally, in the update phase the clique coordinator must consider all reply messages and rearrange the clique accordingly. A new connection can result in the clique merging with another, while a broken connection can cause the clique to split in two. When a connection is formed we designate the coordinator of the largest clique, or of the clique of the selling agent for equally sized cliques, as the new coordinator for the combined clique. Accordingly, the smaller clique's coordinator sends a message to the larger clique's coordinator containing its map. The larger clique's coordinator can then calculate the map for the new combine clique, and send a message to each of its new trader agents informing them of their new clique coordinator. There will be at most $\lceil(\delta S)/2\rceil$ of these, minus one for the previous clique coordinator who already knows of the changes. When two cliques are merged we also upgrade any additional internal matching links into connections. This requires no extra cost.

Also in the update phase, if any traders that were adjacent to a connected link have received a new resource during the turn, that connection must be broken. The clique coordinator can again calculate the new map, and if the clique is split can inform the involved traders, designating one per new clique as the new clique coordinator and giving it its new map. In the worst case the clique will divide into S separate compound agents, thus the clique coordinator must send a maximum of $S-1$ sequential messages. Each of the new clique coordinators can inform their trader agents of the change as part of the messages sent in the next shuffle round.

The total maximum number of message rounds required to achieve all of the above phases is $\lceil(5\delta S)/2\rceil + S - 2$. One way to reduce this cost is to use fixed coordinators within each compound agent as middlemen between the clique coordinators and the trader agents. We can further stipulate that each clique coordinator is also the coordinator for its compound agent. Messages from a clique coordinator to the clique's traders can be sent as single messages to the $S-1$ other compound agent coordinators. The compound agent coordinators then forward them to the $\delta - 1$ other trader agents in the compound agent. Similarly, traders can communicate with their clique coordinator via their compound agent coordinator. Using this optimization reduces the cost of the shuffle and reply phases to $S + \delta - 2$ message rounds each. It also reduces the cost of informing clique members of the new clique coordinator when a clique is merged since only the $\lceil S/2\rceil - 1$ other compound agent coordinators now need to know this information. In all the maximum number of message rounds during a bidding round is reduced to $\lceil(7S)/2\rceil + 2\delta - 4$

The simulations reported in this chapter followed an earlier version of the model presented throughout this work. The simulation results are thus for a slightly different agent

protocol than the one described above (see [33]). In this protocol cliques simply keep track of which ports are adjacent to nonmatching links, instead of storing the addresses of unwanted neighbors. In the shuffle phase cliques give each unmatched port the address of another unmatched port within the clique. Unmatched ports use these addresses to exchange unwanted neighbor addresses. In this protocol fewer messages are required during the reply phase when there are many nonmatching links, though the maximum value remains the same. The exchange phase, on the other hand, becomes slightly more complicated. In the simulations we used an exchange protocol that required 4 messages instead of 2. The simulation results later in this chapter report the actual message round cost for this alternative protocol. The results for the two protocols should, however, be roughly the same. On average, the extra cost of the exchange phase in the old protocol should cancel out the savings made in the reply phase. This is because we use $\delta = 5$, meaning that a compound agent that is part of large clique has at most 4 nonmatching links. On average only half of the traders adjacent to these nonmatching links send a reply message, resulting in the new reply phase costing 2 extra messages. In the simulations, however, we measure the maximum number of message rounds used by any clique during each bidding round. Thus the simulation measurements will be slightly lower with the older protocol. For this reason we also present, for comparison, the message round cost calculated by assuming that each bidding round incurs the maximum cost in the new protocol.

In the next sections we present results from simulations of the peer-to-peer and centralized auctions. We consider for both the number of bidding rounds and the number of message rounds they require to settle at market equilibrium, and later to make subsequent deals. First, however, we look more closely at details of the peer-to-peer auction to determine that it does in fact settle to an equilibrium involving all the trader agents in the system. We then discuss our choice of parameters to use when comparing the two systems. Finally we present this comparison, analyzing how the two systems behave as the number of traders is varied.

4.3 Peer-to-Peer Auction Behavior

Before comparing our peer-to-peer procedure to the centralized approach we must first establish that it produces correct market behavior. As described in Section 4.2, market trade prices should converge over time to a theoretical equilibrium price P_0. In all of our experiments we assign each trader a random reservation price between 100 and 200, creating approximately the supply and demand curves shown in Figure 4.1. Trader agents are also given initial shouts at random, for sellers from the interval $[R_0, 299]$, and for buyers from the interval $[1, R_0]$, where R_0 is the reservation price for that particular trader. We create approximately equal numbers of each trader agent type by assigning each trader to be a buyer or seller at random with a 50% chance of each. P_r, the new resource arrival rate is set to 0.1 following a Poisson distribution. Thus we examine the case in which resources arrive at a steady rate. Since cliques are held together by agents that have made a deal and are waiting for new resources this means different cliques are changing at different times. Uneven resource arrival, on the other hand, would result in periods with more or less traders in the market. In the extream case, if all new resources arrived simultaneously all cliques would break up and reform, probably slowing down the rate at which the market converges. In this

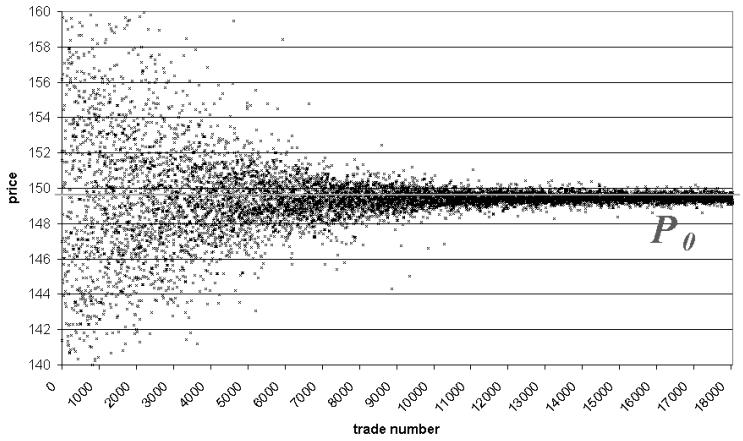


Figure 4.4: Sample trial trade prices.

case, however, we could implement measures to avoid cliques breaking up simultaneously by delaying the time at which connections are broken.

Figure 4.4 shows the series of trade prices produced in a peer-to-peer trail with 2,500 trader agents. We see that these start out widely scattered and over time converge to within 1.3 units of the equilibrium price of 149.6. Smith [43] introduced a way of appraising how close a set of n trade prices p_i are to equilibrium: a measure of the standard deviation of trade prices from the equilibrium trade price, $\alpha = 100 * \left(\sqrt{\left(\sum_{i=1}^{n} (p_i - P_0)^2 \right) / n} \right) / P_0$. Graphing α over time gives us a quantification of how quickly an auction converges to equilibrium, and how closely it matches that equilibrium after convergence. In the following experiments we calculate α for the trades that have occurred every round in our peer-to-peer market and every two rounds in the centralized market. In Smith's experiments with humans, α settled at between 0.6 and 13.2, thus we consider an α value of 1 or 2 to be reasonably low. In Priest and Van Tol's experiments in [37] agents were shown to converge to an α of about 1. We consider two characteristics of the α curves when comparing the two auction types, first the number of bidding rounds it takes to reach a particular α value, and second the average value of α that the market eventually stabilizes on.

Figure 4.5 shows the α curve produced by the trade price data in Figure 4.4. We see that in this trial the auction converges to an α of 1 in 170 bidding rounds and it eventually converges to an average α of 0.17. We believe this shows acceptable market convergence behavior for the peer-to-peer trader agents. We must, however, further consider the extent to which each trader agent is participating in the market. For instance, if only half the trader agents who can trade are trading, the random distribution of reservation prices would still result in the same equilibrium price. Figure 4.6 shows, for a longer trial, the number of trades made by each seller agent, with agents ordered by reservation price. We see that all of the sellers with a reservation price below P_0 do trade. Each make between 30 and 58 trades, independent of how far their reservation price is from P_0. More importantly we see that all traders that should be able to do trade, while there is a steep reduction in the number

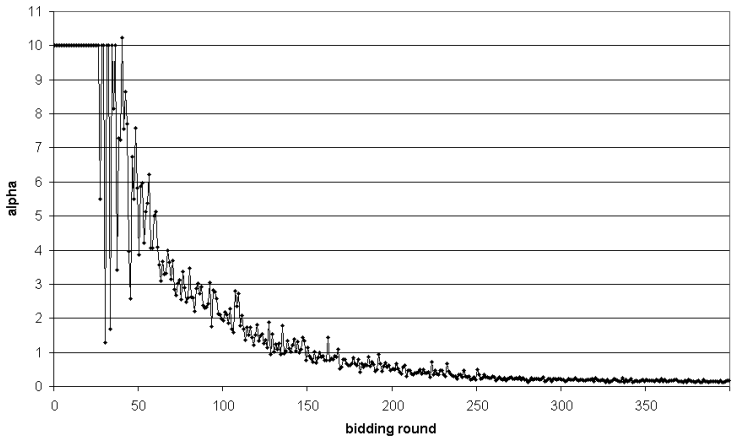


Figure 4.5: Sample scaled standard deviation from the equilibrium trade price, α, over time.

of trades for sellers with a reservation price near P_0, dropping to no trades being made by most agents with R_0 above P_0. Data for the buyers looks similar to that for the sellers.

We also need to inspect the distribution of agents' trading partners. It is possible that trader agents repeatedly make deals with the same partners, so that the system forms fixed cliques rather than changing organization over time. Figures 4.7 and 4.8 show trade-partner data for a sample buyer in a trail in which it made 2,706 trades. Figure 4.7 shows how many trades it made with each of the sellers, ordered by the sellers' reservation prices. We see that the sample buyer traded with almost all of the sellers with a reservation price below P_0. Figure 4.8 shows the distribution of the number of times the buyer traded with each partner, for sellers with a reservation price below P_0. We see that only slightly more than 2% of possible partners were not traded with. For a completely even distribution the buyer agent should have traded with each seller with a reservation price below equilibrium 4.26 times. Indeed we see that the distribution centers near this point. Looking at other trader agents in the system we find similar trade partner distributions.

4.4 Parameter Choices

Having established that the peer-to-peer auction produces acceptable price convergence and that all trader agents participate roughly equally, we can now compare the peer-to-peer auction's performance to that of the centralized auction. In doing so, however, we must consider that performance in both auctions depends upon the parameters used. In each we must consider parameters for the trader agents' price learning algorithm and for the peer-to-peer auction we must look at the clique size limitations as well. Indeed, changes in the parameters lead to tradeoffs among different performance characteristics. Moreover, each market performs well with different parameter sets for the bidding update procedure. In the following section we discuss our choice of the parameters used in our comparison experiments.

Each trader agent's reservation price, buyer or seller status, and initial shout are assigned

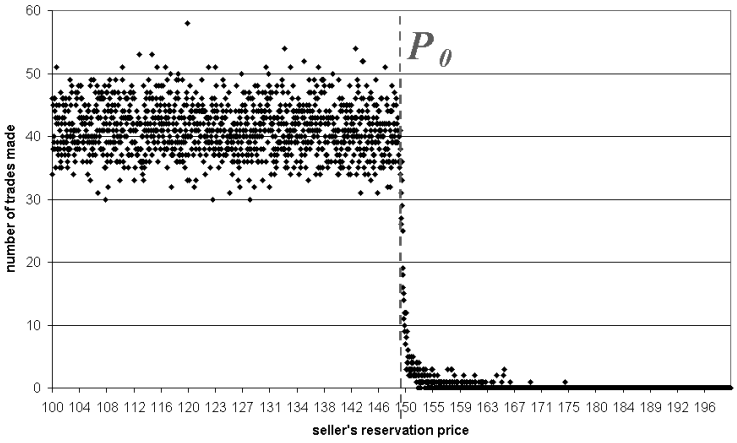


Figure 4.6: Number of trades per seller.

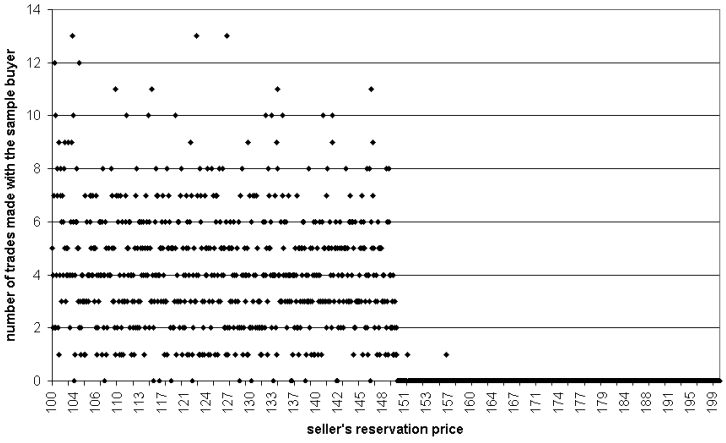


Figure 4.7: Number of trades between a sample buyer and each seller.

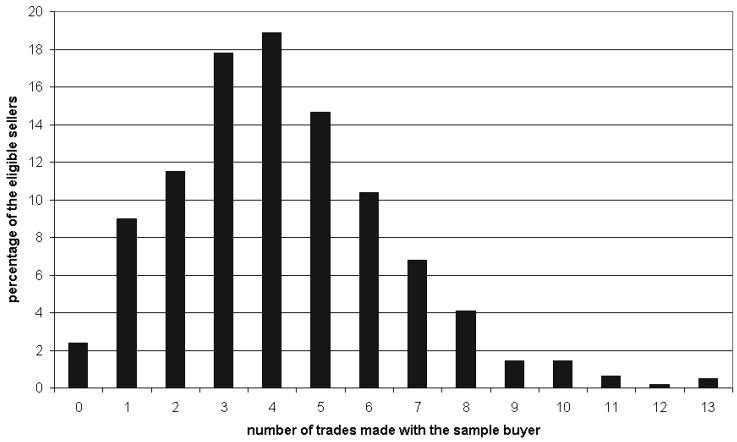


Figure 4.8: Distribution of the number of trades made with eligible sellers in Figure 4.7.

at random in both auctions, as described in Section 4.2. Both auctions have a parameter, P_r, the rate at which new resources arrive. The peer-to-peer auction also contains S, the maximum clique size and δ,the number of agents in a compound-cluster. The new resource arrival rate sets the basic speed at which the auctions run. Both auctions thus need to use the same P_r. For the peer-to-peer auction P_r determines the number of turns for which links will remain matched. Tasks thus need to be replaced at a rate slower than every turn. We therefore set $P_r = 0.1$, a value which we found to work well for the matchmaking procedure. The number of agents in a compound-cluster, δ, and the maximum clique size, S, determine how many message rounds are needed in a clique for each bidding round. For this reason we would like to keep them both small. However, the matchmaking experiments in Chapter 3 show that S and δ also affect how quickly cliques form since they determine how many potential partners each trader has. The more quickly cliques form, the more partners traders see, and the faster trade price convergence occurs. Furthermore, if S or δ are too small compared to the chance of two agents being able to trade, cliques will not form at all. For operations that involve messages between a clique coordinator and clique traders it is most efficient to set S and δ to the same value. Thus we ran experiments with the (S, δ) pairs, (3,3), (4,4), (5,5), (6,6) and (7,7). Figure 4.9 shows α curves averaged over 25 trials at each of these sizes. We see that the convergence rate is improved by increasing (S, δ). However, as S and δ increase, the performance gained by increasing them further becomes smaller. Examining the number of messages rounds until α reaches 1 we find that at the point $S = 5$, $\delta = 5$ the increase in convergence rate no longer makes up for the greater number of message rounds per bidding round. Thus in the following experiments we use the values $S = 5$, $\delta = 5$.

For both auctions the trader agent bid update algorithm has the following parameters, all of which affect the rate at which a trader changes its asking price: momentum (γ), the extent to which price changes are based on previous price changes; learning rate (β), how quickly a trader moves towards its current target price; R_1, the upper limit for a random variation based

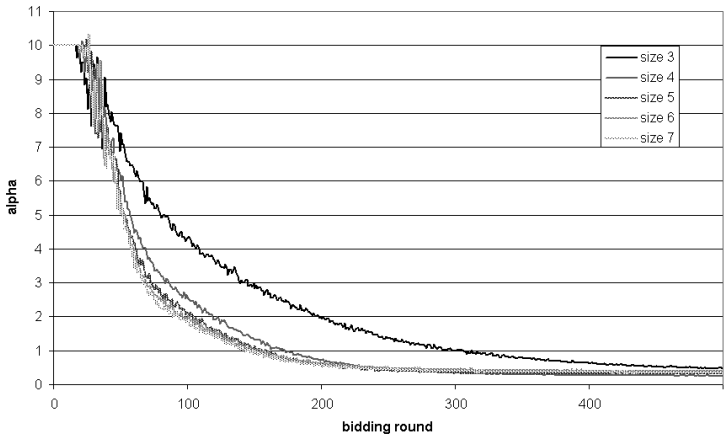


Figure 4.9: Average α curves, varying clique sizes.

on current price; and R_2, the upper limit for a random variation independent of the current price which ensures that prices always move a bit. Of these we found that the momentum made the most difference to performance. Momentum determines how willing a trader is to change its price more or less than it did the turn before. With dependable information, as in the central auction, momentum can be low, as the traders can be fairly sure that their target price represents the market as a whole. On the other hand in the peer-to-peer model traders are given very low quality information and thus need a high momentum, meaning that they base their price changes more on the history of target prices than on just the current one. Based on this analysis, and some exploration done by hand, we choose momentum values for the two auctions; $\gamma = 0.05$ for the centralized auction, and $\gamma = 0.9$ for the peer-to-peer auction. From this point we used more hand tuning and a genetic algorithm search to determine good values for the other bid update parameters. Our aim in determining these parameters was to obtain the fastest possible dependable convergence while keeping the α values after convergence below 1. These two goals are often at odds, traders that change prices quickly will converge more quickly, but are also more likely to jump to prices further from equilibrium once convergence has occurred. For instance, increasing the learning rate, which determines how much traders are willing to move each turn towards their opponent's price, decreases the time to convergence. If the learning rate is set too high, however, values for α vary greatly after convergence. Similarly with R_1 and R_2, which create some randomness that is used to keep the market moving. High values for R_1 and R_2 lead to faster convergence, but lowering their values creates a more stable end solution. We confirmed that the values used in [37] produce good behavior in the centralized auction, and thus use the same values in our comparison experiments: $\gamma = 0.05$, $\beta = 0.3$, $R_1 = 0.2$, $R_2 = 0.2$. We found that the high momentum used in the peer-to-peer auction resulted in a fairly unstable α after convergence, and to counteract this effect we lowered the values for the learning rate, R_1 and R_2. The resulting parameters for the peer-to-peer auction were thus: $\gamma = 0.9$, $\beta = 0.25$, $R_1 = 0.001$, $R_2 = 0.02$.

CENTRALIZED SYSTEM DATA											
	bid. rounds to α=2.12			message rounds to α=2.12			bid. rounds / trader deal	mes. rounds / trader deal	final α value		
num. traders	min	avg	max	min	avg	max	avg	avg	min	avg	max
2,500	36	53.72	88	167326	246513	399055	19.53	88343	0.05	0.40	2.02
5,000	38	54.38	76	351416	498646	687750	19.65	177974	0.03	0.34	1.16
10,000	42	55.00	66	776057	1009479	1205547	19.87	360268	0.03	0.31	1.18
20,000	46	55.74	68	1696591	2046734	2482085	20.02	726712	0.02	0.33	0.95
40,000	48	55.12	70	3537399	4048187	5107476	20.15	1464140	0.01	0.29	0.72
80,000	50	55.16	64	7360688	8104125	9389452	20.27	2947772	0.02	0.30	0.63
160,000	52	55.77	60	15305138	16404413	17636285	20.37	5929863	0.05	0.29	0.53
PEER-TO-PEER SYSTEM DATA (MEASUREMENT FROM SIMULATION)											
2,500	102	127.16	224	1529	2038	3983	22.30	447	0.05	0.39	1.62
5,000	97	117.77	147	1486	1914	2516	22.30	461	0.07	0.31	1.02
10,000	100	109.08	130	1565	1788	2226	22.29	473	0.08	0.22	0.62
20,000	95	105.02	120	1529	1747	2056	22.34	485	0.08	0.20	0.52
40,000	95	101.60	113	1565	1711	1968	22.29	494	0.09	0.15	0.40
80,000	96	100.75	109	1628	1729	1922	22.64	509	0.11	0.17	0.37
160,000	97	101.17	105	1679	1770	1868	22.62	517	0.11	0.16	0.25
PEER-TO-PEER SYSTEM DATA (CALCULATED MAXIMUM COST FOR NEW PROTOCOL)											
2500				2448	3052	5376		535			
5000				2328	2826	3528		535			
10000				2400	2618	3120		535			
20000				2280	2520	2880		536			
40000				2280	2438	2712		535			
80000				2304	2418	2616		543			
160000				2328	2428	2520		543			
HIERARCHICAL CENTRALIZED SYSTEM ESTIMATIONS											
2,500				1531	2285	3743		830			
5,000				1759	2518	3519		910			
10,000				2103	2753	3304		994			
20,000				2476	3001	3661		1077			
40,000				2765	3175	4032		1160			
80,000				3068	3385	3928		1244			
160,000				33879	3633	3908		1327			

Table 4.1: experimental data summary

4.5 Comparison of the Two Auctions

We now investigate how the behavior in the peer-to-peer and centralized auctions changes as we increase the number of trader agents participating. For each market type we ran 100 trials with 2,500, 5,000, 10,000, 20,000, 40,000, 80,000 and 160,000 trader agents using the parameters chosen in Section 4.4. A summary of the data from these trials is given in Table 4.1. In this section we consider more closely the number of bidding rounds it takes the auctions to reach equilibrium, the number of message rounds this involves, and after equilibrium is established, how many message rounds are needed to make each subsequent deal.

Figure 4.10 shows the average α curves for the centralized and peer-to-peer auctions with 2,500, 10,000 and 40,000 trader agents. We find that the centralized auction converges roughly two times faster. We also find that as the number of trader agents is increased the rate of convergence in both markets remains approximately the same. The last three columns of Table 4.1 show minimum, average and maximum α values after convergence, averaged over the last 100 bidding rounds of each trial. We find that trials with more trader agents are more exact; as more trader agents are added the spread of final α values, and more importantly the maximum final α value decreases. This creates complications when picking an α value at which to measure the time to convergence. Over all the trials the highest α after convergence was 2.12 and thus we chose to measure convergence as the number of rounds taken to reach an α of 2.12. This is high for the trials with many trader agents but makes the points with fewer trader agents easier to read. Using an α of 1 as we did earlier gives similar results but makes interpreting points with fewer agents less clear-cut. From Table 4.1 columns 2 to 4 we can see that the number of bidding rounds to α=2.12 remains almost constant as the number of trader agents increases, at an average of about 100

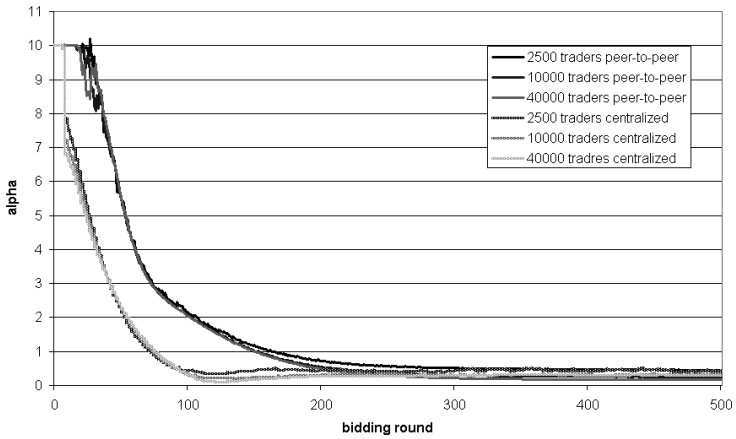


Figure 4.10: Average α curves, varying system sizes.

bidding rounds in the peer-to-peer auction and about 55 bidding rounds in the centralized auction. In the peer-to-peer case the average number of bidding rounds decreases a little as the maximum value drops towards the minimum.

For an auction running over a network it is likely that communication costs will make a large contribution to the actual time taken by the bidding rounds plotted in Figure 4.10. Thus we also consider the number of message rounds that occur within the auctions up to our convergence point. For the centralized auction we count the message rounds required by the auctioneer per trading round as discussed in section 2.1. For the peer-to-peer auction we count the maximum number of message rounds required by any clique in a trading round, as discussed in Section 4.2.2. We sum these for the bidding rounds up to and including the first trading round where α is below 2.12, giving the results in columns 5 to 7 of Table 4.1. We see that the number of message rounds required to reach equilibrium in the centralized auction increases linearly with the number of trader agents, as the auctioneer has to deal with increasingly more bids. In the peer-to-peer auction, on the other hand, it remains pretty much constant since the maximum size of the cliques does not change. For the large system sizes tested in our experiments the peer-to-peer auction always outperformed the centralized auction on this measure. Extrapolating from the data the two systems should be equal on this measure at around 165 trader agents.

Finally, Table 4.1 columns 8 and 9 also presents data on the time trader agents must wait between each deal they make. This becomes more important than the cost of converging in long running auctions where the supply and demand curves do not change quickly. The data shown was determined by counting the message rounds and deals during the last 100 bidding rounds, averaged over 100 trials. We estimated that half the trader agents in each trial, those with a reservation price that allowed them to trade, were making these deals. Column 8 shows that for both the peer-to-peer and centralized auction the average number of bidding rounds it takes a trader to make a deal remains approximately constant. Column 9 however shows that when considering the message rounds the trading time in the centralized

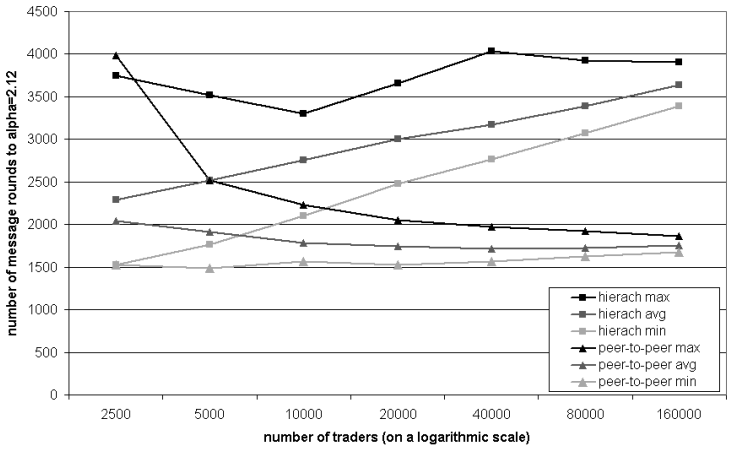


Figure 4.11: Message rounds to α=2.12.

auction increases linearly. For the peer-to-peer auction we expect a constant cost, however we also see a small increase. There is a maximum of 26 message rounds per bidding rounds given our values of $S = 5$, $\delta = 5$. This is a maximum value, however, most clique's bidding rounds take less time. The increase in message rounds in column 9 occurs because the average number of message rounds per trading round increases from about 20 to 23 as the number of cliques is increased. However, provided that the number of bidding rounds per trader agent deal remains constant at about 23 the number of message rounds per trader agent deal will be capped at 598.

Our experiments considered a simple centralized auctioneer. In a large system, however, the benefits of distributing this auctioneer outweigh the costs. Such a distributed central auctioneer could be realized, for instance, by creating a hierarchical tree of auctioneer nodes in place of a single node that must handle all messages. In this case the leaf nodes receive messages from the traders, combine them, and send them to the next level, and so forth up to the root, with the reverse procedure for return messages. The optimal branching factor for such a tree is e. Each bidding round in the auction involves a message from all the traders to the auctioneer, and one in return from the auctioneer to each trader. Thus in Table 4.1 for the hierarchical central system estimation (columns 5, 6, 7 and 9), we estimate the message rounds per trading round with such a parallel structure as $2e\ln(N)$ and multiply this by the number of bidding rounds measured in our simulations of the simple centralized system. Figure 4.11 compares the resulting message round to $\alpha = 2.12$ costs to the peer-to-peer auction. We see a great improvement over the simple single central auctioneer, the distributed auctioneer produces only a logarithmic increase in costs, however we find that the peer-to-peer auction still outperforms the centralized version for auctions with more than around 15,000 agents. We further find in Table 4.1 column 9 that the message rounds per deal after equilibrium also increases logarithmically with a distributed auctioneer. This measure still remains higher than in the peer-to-peer system.

We did not measure memory and processing costs in our experiments but we can ar-

gue that for any individual entity the comparisons between the peer-to-peer and centralized auctions are similar to those of the message costs. In the centralized auction the auctioneer incurs both high memory and high processing costs, while in the peer-to-peer auction costs are distributed among all the agents. The central auctioneer stores bids and offers from all traders at each round, costing $O(N)$ memory. Further to calculate deals it needs to sort the lists of bids and offers, costing $O(N \log N)$ time, to find the best ones. As for messages, creating a hierarchical auctioneer reduces this sorting cost to $O(\log N)$. Meanwhile, in the peer-to-peer auction the limitations on the number of traders in a compound agent and clique sizes keep memory and processing requirements per turn at the clique coordinators (the agents that do most of the work) constant.

4.6 Conclusions

This chapter demonstrated how the peer-to-peer agent matchmaking procedure could be used to implement a peer-to-peer agent auction. We showed that in spite of the fact that peer-to-peer trader agents each know only a limited amount of local information, such an auction is able to exhibit market-price convergence. Through experiments with a particular agent bidding algorithm and an example supply-and-demand curve we showed that this peer-to-peer auction has a constant cost in the number of bidding rounds it takes to find equilibrium from a random starting point. While this cost was about two times higher than for a comparison centralized auction where an auctioneer distributes global information, we showed that the cost of reaching equilibrium in terms of message rounds was far better in the peer-to-peer case. For the peer-to-peer auction this message rounds cost remains constant as the number of agents increases while for the centralized case it increases linearly. Even when compared with a distributed hierarchical auctioneer, the peer-to-peer auction showed better performance for systems with 15,000 or more agents. The exact numbers presented in this chapter depend a great deal upon the bidding strategy of the agents that we chose for our simulations. Changing the bidding strategy should, however, have a proportionally equal effect on the number of bidding rounds needed to reach market convergence in both the peer-to-peer and centralized auctions. The difference in structure between the two auctions, on the other hand, creates a fundamental difference in the growth rates of the costs of communications as the number of traders increases, which should not change as the individual strategies of the agents change.

Chapter 5

Grouping Matchmaking Agents

The previous chapters studied agents that sought separate matches for independent tasks. Because all tasks were picked at random these agents were, in effect, identical. More realistically, agent attributes, which in the general model describe agents' characteristics, will be heterogeneous, and agent's objectives are likely to relate to their attribute. In this setting agents, rather then being able to form matches equally well with any other agents, will each have a set of other related agents with which they are more likely to get along. A good matching of the entire system should then place certain agents together. As such a configuration represents a form of system organization, discovering it is a more difficult undertaking than finding the simple individual pairings that have been studied so far.

In this chapter we thus concern ourselves not just with finding matching links between agents, but also with behaviors that can enable agents, and cliques, to find matches that are among the best (i.e. strongest) ones available. We further introduce agent objectives (tasks) that are chosen as a function of an assigned agent attribute. We run experiments in which an initial matching is formed then modified by replacing objectives, thus updating the matching and connecting link sets. Our goal is for these agents with heterogeneous attributes to become linked together in a configuration in which agents within a clique have similar attributes.

5.1 Model

We use the final matchmaking agents from Chapter 3 as the starting point for our experiments. Figure 5.1 gives the pseudo-code for these agents. This pseudo-code is basically the same as that for the agents in Chapter 3. In this chapter, however, instead of considering all agents to be identical we assign to each agent a different attribute (line 15), which is simply a Real value. Instead of "tasks" agents have abstract objectives, which are also simply Real values (line 2). Both attributes and objectives are chosen out of the interval $[0,M)$, and as in Section 3.5 we consider this interval to be cyclic when measuring distances between objectives and attributes. We define a function, $p_{type}(d)$, where d is the distance to an agent's attribute, which is used to assign objective values (line 5). We also change the manner in which we end a match between two objectives, from being based on a fixed probability, P_e, to a probability function $p_{end}(d)$, where d is the distance between the matched objectives (line 52). This allows us to distinguish between weaker and stronger links when downgrading links.

In the following experiments we study how clique composition evolves when agents select certain links to maintain as matching or connecting links and reject others. Conceptually, agents "end" matched objectives, downgrading the associated link, when that link is no longer useful. Thus, as in Section 3.6, objectives of the ports adjacent to a link are replaced when the link is downgraded. Accordingly, when *reconsiderComposition* chooses to end a matching or connecting link, based on p_{end} (line 49 and 50), the link is downgraded to nonmatching (line 4), and its adjacent ports are given new objectives (lines 5 and 6).

5.2 Adjusting Agent Functions

In this section we study methods by which matches between objectives (as represented by matching and connecting links) and agent groupings (as represented by cliques) can be improved by making changes only to the agents' individual behaviors. Further improvements to cliques are likely to require collective actions among the clique members, but as these procedures require communication and agreement between agents, we first examine what degree of grouping can be achieved in their absence.

5.2.1 Improving the Strength of Matching and Connecting Links

One way to improve the quality of matching and connecting links is to modify agents so that they favor stronger links when making matches and connections, and more readily downgrade weaker links. For our current agents, strong links are those that have a short length, where a link's length is the distance between the objectives held by the two ports adjacent to the link. The basic agents in Chapter 3 did not make a distinction between weaker and stronger matching, or connecting, links. Figure 5.2 shows the distribution of the lengths of matching and connecting links found by these initial agents, described in detail below. It shows that all possible link lengths (those within the support of the matching probability function) are equally likely. We would like to develop agents that instead favor maintaining links with shorter lengths.

For the initial case in these experiments we consider agents that do not take account of their attribute when choosing objectives. Instead, objectives are chosen uniformly at random out of $[0, M)$. We choose values for agent parameters that are well within the range in which cliques form. We set the maximum clique size to S=30, since previously we had found that size limits between 10 and 100 produce fairly similar results in terms of trials connecting or not connecting. A smaller S could be used if we want to limit clique computation costs, while a larger S could be chosen if we wish to divide the system into fewer units. Initially we use the rectangular match probability function (with $d_{max} = 0.5$), from Section 3.5. We choose matching and connecting links to downgrade according to the function, $p_{end}(d)$ = 0.003. We found earlier that this changes the matching and connecting link set slowly enough to avoid matched objectives ending before cliques have a chance to form, yet fast enough that cliques will continue to evolve once they have formed. $p_{end}(d)$ could be set higher if we wish to see changes happen in fewer turns. In summary, the parameters used for Figure 5.2 are:

- N = 10,000; M = 60; δ = 3; S = 30

```
1.  object Port{
2.     objective: Real; neighbor:Port; matching:Boolean; connecting:Boolean; myAgent:Agent;
3.    internal procedure reset(){
4.      connecting := FALSE; matching := FALSE;
5.      objective := ``select new objective according to p_type'';
6.      neighbor.reset();
7.    }
8.    internal procedure  matches( p2:Port ) returns ( Boolean ){
9.      return(``true with probability p_match(d)'';
      //Where d is the distance between objective and p2.objective, as defined in Section 3.5.
10.   }
11. }
12. object Agent{
13.    acquaintances: internal set of Port; //δ links per agent
14.    c: Clique; //The clique of which this agent is a member
15.    attribute: Real;
16.    process searchForMatches [for each aq in acquaintances]{
17.      whenever ( ``clock phase 3''  & aq.matching = FALSE & aq.connecting = FALSE ){
18.        aq.neighbor := c.tradeIn( aq.neighbor );
19.        aq.matching := aq.matches( aq.neighbor )
20.    }
21. }
22. virtual object Clique {
23.    members: set of Agent; //All agents that are members of this clique
24.    unwantedAcquaintances: set of Port;
25.    external procedure  tradeIn( p:Port ) returns ( p':Port ){
26.      unwantedAcquaintances.add( p );
27.      wait until( ``clock phase 2'' );
28.      return  unwantedAcquaintances.remove( ``random element'' );
29.    }
30.    process searchForConnections [for each m in members]{
31.      whenever  (``clock phase 1'' ){
32.        for each  aq in m.acquaintances {
33.          if ( aq.matching = TRUE & aq.connecting = FALSE ){
34.            if ( aq.neighbor.myAgent.c.requestConnection( self ) = ACCEPT ){
35.              aq.connecting := TRUE;
36.              ``join the cliques'';
37.            }
38.          }
39.        }
40.      }
41.    }
45.    external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when (``available''){
46.      if (members.size + requester.members.size ≤ S ) return ACCEPT;
47.      else return REFUSE;
48.    }
49.    process reconsiderCompositon() [for each m in members]{
50.      whenever  (``clock phase 4''){
51.        for each  aq in m.acquaintances {
52.          if ( aq.matching = TRUE & ``true with probability p_end(d)''  ){
         //Where d is the distance between aq.objective and aq.neighbor.objective
53.            aq.reset();
54.            ``split off cliques if necessary'';
55.          }
56.        }
57.      }
58. }
```

Figure 5.1: Pseudo-code for matchmaking agents with heterogeneous attributes.

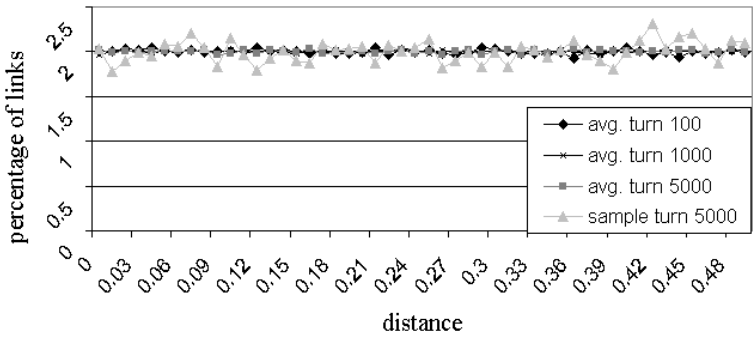


Figure 5.2: Distribution of distances between matched objectives for initial agents; p_{match_0}, p_{end_0} (see text).

- objectives are assigned uniformly at random from M, i.e. $p_{type}(d) = 1/M$ when $d < M/2$ and 0 otherwise.
- $p_{match_0}(d) = \begin{cases} 1, & \text{if } d < 0.5 \\ 0, & \text{otherwise.} \end{cases}$
- $p_{end_0}(d) = 0.003$.

We ran 100 trials, each for 5000 turns. Figure 5.2 shows the distribution of distances between matched objectives measured at 100 and 5000 turns over all trials, and at 5000 turns for a sample trial. As links are upgraded and downgraded with equal probability over the whole range of distances between 0 and 0.5 we expect to see an even distribution that does not change over time. Our objective is to shift the mean of this distribution towards 0. In principle, this could be done by decreasing the support of p_{match}. As we saw before, however, doing this would decrease the maximum possible M for which cliques would still form. Instead, we consider two alternative methods that allow a large range of matching links to form, but maintain only the better ones. First, we consider increasing the probability of stronger matching links being formed by using the triangular match probability function from Section 3.5:

$$p_{match_1}(d) = \begin{cases} 1-2d, & \text{if } d < 0.5 \\ 0, & \text{otherwise.} \end{cases}$$

Second, we consider decreasing the probability that strong matching or connecting links will be downgraded by using a triangular-shaped ending function in which links of length 0 have the least chance of being downgraded, and links of length 0.5 or higher have the greatest chance of being downgraded. We set an ending probability of 0.001 at distance 0 (we don't use 0 since we want all objective matches to end eventually) and increase linearly to a probability of 0.005 at $d = 0.5$, to give the same average probability of downgrading a matching or connecting link (among those that can possibly be formed) of 0.003 as in the flat distribution used before.

$$p_{end_1}(d) = \begin{cases} 0.008 * d + 0.001, & \text{if } d < 0.5 \\ 0.005, & \text{otherwise.} \end{cases}$$

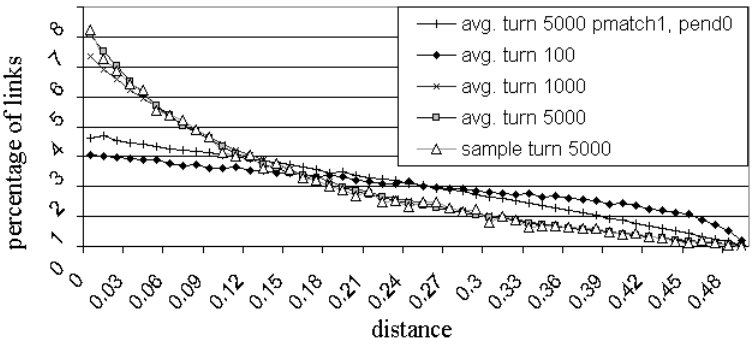


Figure 5.3: Distribution of distances between matched objectives for initial agents; p_{match_1}, p_{end_1}.

Figure 5.3 shows the resulting distribution of distances between matched objectives. The first data set shows the end result of using p_{match_1} and p_{end_0}. The other data sets are for p_{match_1} and p_{end_1} at various times during a trial. We see that with the combination of both selective functions the distribution of match distances improves over time, to give the configuration we hoped to achieve, in which shorter matching and connecting links are much more likely to occur.

5.2.2 Basing Objective Values on Agent Attributes

Creating short matching and connecting links, on its own, does not imply that agents within a clique will be closely related to each other. If we assign to each agent a Real value attribute we can measure not only the distance between matched objectives, but also the distance between agents within the same clique. We can then use the average distance between the attributes of all agents in a clique as a measure of the clique strength, or the relatedness of a clique's component agents. For this strength measure we use the term *clique density*[1]. Figure 5.4 shows a base case for this measure in which we give to the current agents, from Section 5.2.1, an attribute value chosen uniformly at random out of $[0,M)$. The objectives of these agents are independent of their attributes, so this graph represents cliques containing wholly unrelated agents. Since the agents use $M = 60$, the expected distance between two random agents is 15, and indeed the clique densities are near to 15. We show curves at different points in time to show that the cohesiveness of cliques remains constant. Our objective is to develop agents for which clique density improves over time, moving the mean of the distribution towards 0. There is one problem, however, with this measure of clique strength; cliques of size 1 have an average distance of 0 between their agent. Thus we see a spike at 0 in the curve which lowers over time as cliques form. To avoid this anomaly we also consider data for only the larger cliques that contain 15 or more agents.

Figure 5.5 shows an alternate measure of clique strength, which we define as the *clique covering*. Here we measure how much of $[0,M)$ a clique "covers" adjusted for clique size. For a clique, c, of size $|c|$ each agent in the clique is considered to cover an interval of size $M/|c|$ centered at its attribute value. We measure the size of the largest interval not covered

[1]Note that this term is somewhat counterintuitive, lower clique density values indicate more compact cliques.

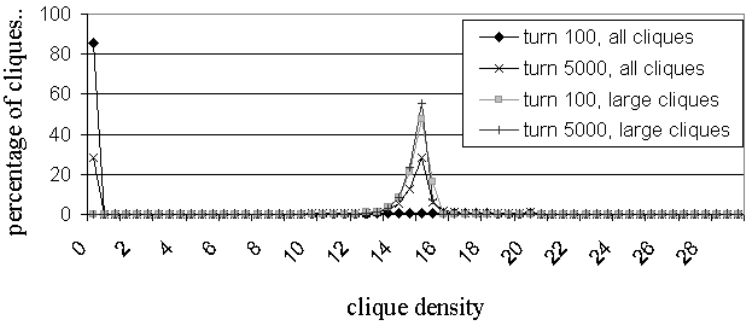


Figure 5.4: Clique density distribution for unrelated cliques; p_{match_1}, p_{end_1}, random objectives.

in this way by any agents in the clique. We plot M minus this amount. Thus a clique of 1 agent covers the whole of the space and has a covering value of 60, a clique of 2 agents with the same attribute covers half the space and has a covering value of 30, etc. Our objective is to move this value towards 0. For a clique size of 30 the minimum value is 60/30=2. Again we show values recorded at 100, 1000 and 5000 turns for all cliques, and only for cliques with 15 or more agents. We again see that in the current trials agents in a clique are unrelated and thus covering values are near the maximum value of 60, with a spike for single agent cliques (at 60).

Limiting the objective values an agent can have to be within a certain distance of its attribute value should result in stronger cliques. To do this we make the following extension to the model: Each agent a in A is given one attribute r belonging to a category in $[0,M)$. Attributes are assigned uniformly at random from $[0,M)$. An agent's objective values are assigned according to a type function $p_{type} : T \rightarrow T$ where $p_{type}(d)$ is the probability of an agent with an attribute of category t having an objective of category t' at a distance d from t.

For our initial p_{type} we allow objective values for an agent with attribute x to be chosen from the interval $[x-10, x+10]$. We choose such a large interval, of size 20, or 1/3 of M, to avoid agents being so specialized that clique formation is prevented. We will later explore what the minimum size of this interval can be, but as this should depend on M, δ, S and the support of p_{match}, the choice may not be straightforward.

$$p_{type_1}(d) = \begin{cases} 1/20, & \text{if } -10 < d < 10 \\ 0, & \text{otherwise.} \end{cases}$$

Figures 5.6 and 5.7 indicate that, to some degree, cliques of agents with similar attributes are formed when objectives are related to their agent's attribute in this way. On the other hand, these cliques are far from having clique densities near 0 or clique coverings near 2, which we would find in a system that was sorting agents into cliques based on attribute values. We therefore attempt to improve clique strengths by increasing the probability of an agent having objectives closer to its attribute value. We use the function:

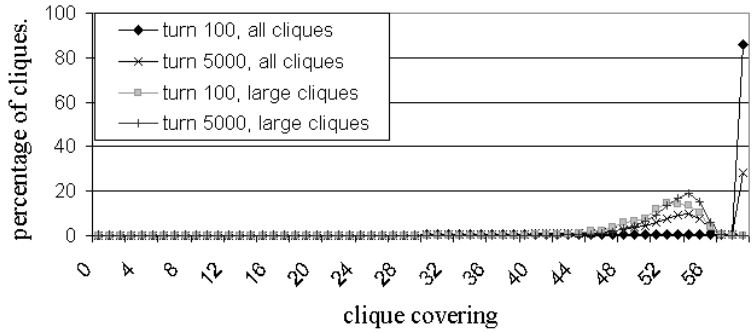


Figure 5.5: Clique covering distribution for unrelated cliques; p_{match_1}, p_{end_1}, random objectives.

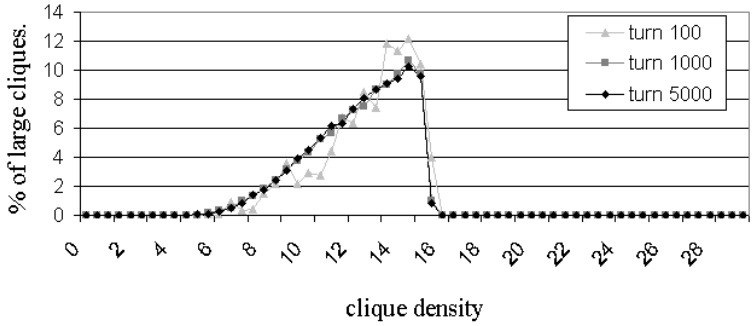


Figure 5.6: Clique density distribution; p_{match_1}, p_{end_1}, p_{type_1}.

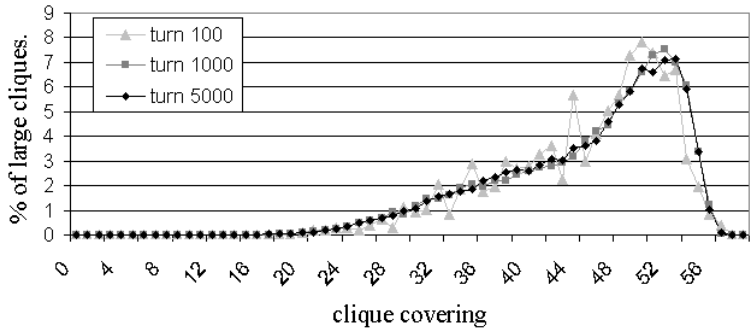


Figure 5.7: Clique covering distribution; p_{match_1}, p_{end_1}, p_{type_1}.

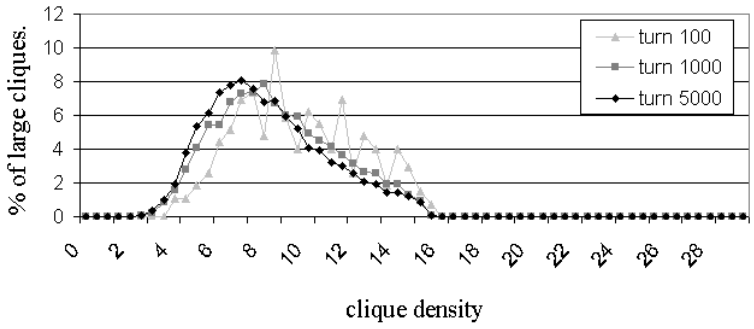


Figure 5.8: Clique density distribution; p_{match_1}, p_{end_1}, p_{type_2}.

$$p_{type_2}(d) = \begin{cases} -\frac{1}{20}\log\frac{d}{10}, & \text{if } 0 < d < 10 \\ -\frac{1}{20}\log\frac{-d}{10}, & \text{if } -10 < d < 0 \\ 0, & \text{otherwise,} \end{cases}$$

which has the same support as p_{type_1} but greatly increases the probability of shorter attribute-objective distances, d. Figure 5.8 and Figure 5.9 show a good improvement in clique strengths. Using p_{type_2}, most cliques are stronger than they were in the random case.

5.2.3 How Selective Can Agents Be in Choosing Objectives?

Reducing the maximum distance between an agent's attribute value and its objective values should increase the strength of cliques. On the other hand, along with the maximum distance at which matches can be made, the possible distance between an agent's attribute and objectives influences the system's ability to form cliques, for a fixed M. To counterbalance this effect, δ, the number of objectives per agent, can be increased. To explore this we run a series of experiments in which the maximum distance between matching objectives is fixed to 0.5, using p_{match_1} and p_{end_1}, but the maximum distance between an agent's attribute and objective is varied using the type probability function:

$$p_{type_3}(d) = \begin{cases} -\frac{1}{r}\log\frac{2d}{r}, & \text{if } 0 < d < r/2 \\ -\frac{1}{r}\log\frac{-2d}{r}, & \text{if } -r/2 < d < 0 \\ 0, & \text{otherwise,} \end{cases}$$

where r is the size of the interval of possible objective values for an agent (r was 20 in Section 5.2.2). We increase r from 2 to 20 by increments of 2, running 10 trials for 5,000 turns at each value. We repeat this experiment for agents with differing numbers of ports, using $\delta = 3$, 4 and 5 and examine the following:

- The minimum r for which all trials are connecting. Figure 5.10 shows for each value of δ the number of trials (out of 10) that formed large cliques. For $\delta = 3$ the minimum r at which all trials are connecting is 14, i.e. an agent with attribute x can have objectives values in the range $[x-7, x+7]$. Each agent thus has objectives from about

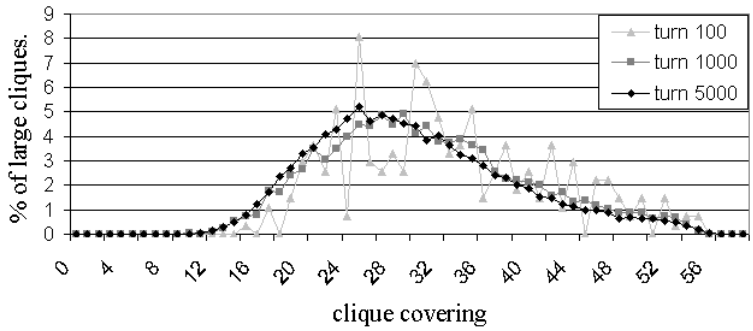


Figure 5.9: Clique covering distribution; p_{match_1}, p_{end_1}, p_{type_2}.

23% of the total category space. At $\delta = 5$, r can be as low as 4, or 7% of the category space. Increasing δ further will probably allow us to continue to decrease r, though the improvement appears to lessen as δ increases.

- The distribution of clique densities at turn 5000 for cliques of size 15 or larger over all connecting trails. Figure 5.11 shows that decreasing r does improve the clique density. This can be seen most clearly for $\delta = 5$, for $r = 4$ clique density ranges from 0.5 to 4.5 centering around 1.

- The distribution of clique coverings at turn 5000 for cliques of size 15 or larger, over all connecting trials. Figure 5.12 again shows the clique strength improving as r is reduced. For $\delta = 5$, $r = 4$ we find some cliques with a covering value as a low as 4, or 7% of the total range, though the distribution centers around 7, about 12% of the total range.

5.3 Clique Centric Decision Functions

In the previous sections of this chapter we showed how the composition of cliques can be changed by changing agent behaviors. In this section we attempt to adjust functionality at the clique level to further improve clique strength. We regard links to be between agents instead of objectives and base a link's strength on the distance between the attributes of the agents it joins. We further consider making and downgrading connections based on the best links available to the clique as a whole.

As in the previous experiments in this chapter, we measure our success in getting agents to group by plotting clique covering. We leave out clique density as it gives roughly the same information as the clique covering. Instead, we look more closely at clique composition by picking one attribute value, 30, and determining what percentage of the cliques contain agents with an attribute value range that includes this value.

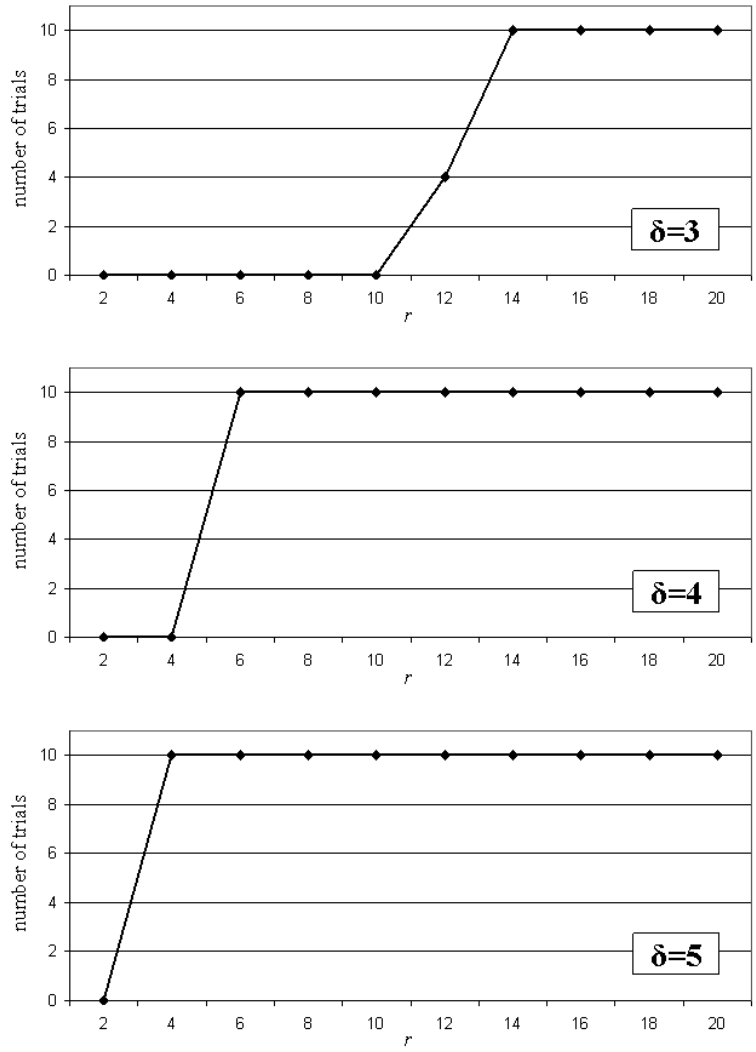


Figure 5.10: Number of connecting trials out of 10.

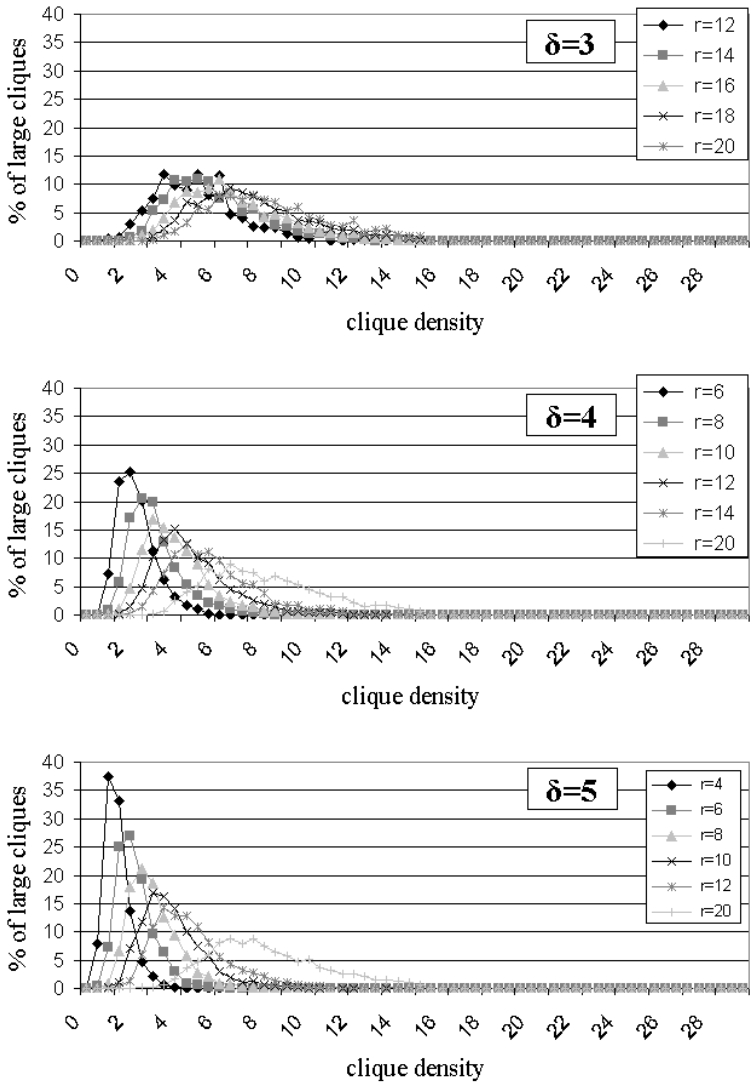


Figure 5.11: Clique density distribution.

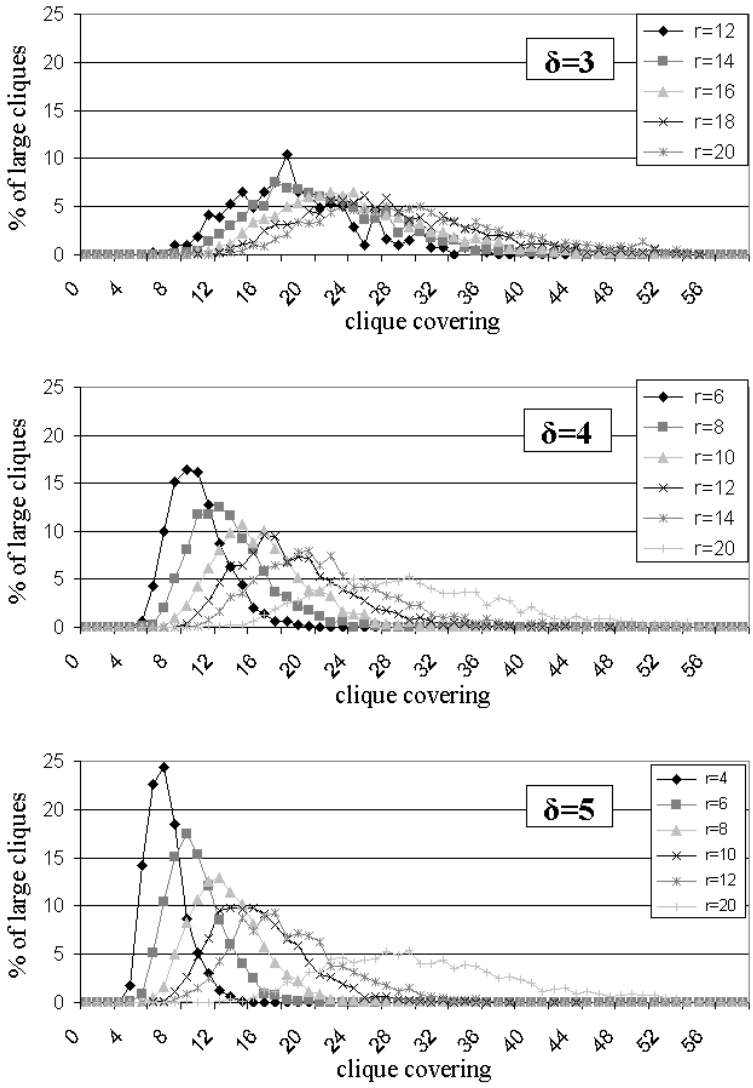


Figure 5.12: Clique covering distribution.

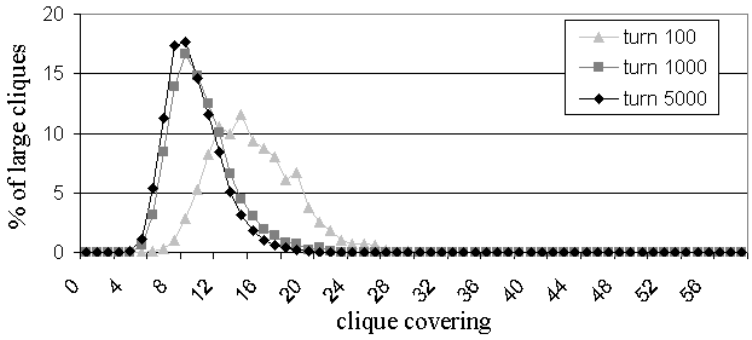


Figure 5.13: Clique covering distribution, agent decisions only; p_{match_2}, p_{end_2}, $t = 0.5$, p_{type_3}, $r = 6$.

5.3.1 Replacing Objectives Based on Agent-Agent Distance

For comparison, we begin with a base case of values picked from Section 5.2.3, that exhibited the best agent centric-groupings achieved. In order to allow us to adjust the possible lengths of matching links, we create a new parameter, t, which defines the maximum distance between matching objectives. Accordingly we also define new versions of the matching and ending probability functions.

- N = 5,000; M = 60; δ = 5; S = 30, $t = 0.5$
- $p_{type_3}(d)$ with r=6
- $p_{match_2}(d) = \begin{cases} 1 - d/t, & \text{if } d < t \\ 0, & \text{otherwise.} \end{cases}$
- $p_{end_2}(d) = \begin{cases} 0.004 * d/t + 0.001, & \text{if } d < t \\ 0.005, & \text{otherwise.} \end{cases}$

We use an initial value of $t = 0.5$, corresponding to the fixed value used in p_{match_1} and p_{end_1}.

We run 100 trials for 5000 turns. Figure 5.13 shows the distribution of clique covering values at various times. In turn 5000 this centers at about 9 with a lowest value of 4. Ideally we would like to shift this distribution further towards the minimum value of 2.

To get a better idea of exactly how well the cliques formed in this experiment partition the data set, we examine the percentage of cliques that would have to be searched to find an agent with a particular attribute value. Upon examining the existing cliques in turn 5000 of a sample trial, we find that out of a total of 208 cliques, 23, or 14.4%, have an attribute value range that contained the value 30. Figure 5.14 shows for these 23 cliques the attribute values for each agent member. Each line in the graph represents one clique, with its members ordered by attribute value. Over all 100 trials we find a minimum, average and maximum values of 9.8%, 12% and 15% respectively for the percentage of cliques with an attribute range crossing the value 30.

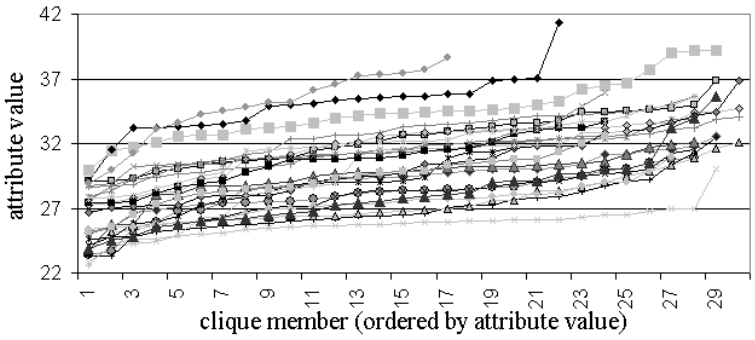


Figure 5.14: Cliques whose attribute range crosses 30, agent decisions only; p_{match_2}, p_{end_2}, $t = 0.5$, p_{type_3}, $r = 6$.

Clique strength is a function of the attributes of the agents in the clique, yet until now we have created and downgraded matching and connecting links based only on objective values. We next attempt to improve clique strengths by basing the strength of connecting links on the distance between the attributes of agents instead of on the distance between the objectives adjacent to the link. The strength of matching links remains based on objective values. To achieve this, we keep the same end probability function, but use the distance between the attribute values of the agents adjacent to a link, instead of the objective values of the ports adjacent to the link. Thus agents can be thought of as forming collaborations based on a current objectives, but keeping friends based on how similar agents are as a whole. For an objective interval, r, of size 6, the objectives of an agent can be at most 3 units distant from that agent's attribute. Thus with a maximum matching distance, t, of 0.5 the maximum distance between the attributes of two agents adjacent to a matching link is 6.5. We therefore introduce a variable $v = 6.5$ to define the shape of the end probability function. We run trials with the same parameters as for Figure 5.13 except for we use this new end probability function:

$$p_{end_3}(d) = \begin{cases} 0.004 * d/v + 0.001, & \text{if } d < v \\ 0.005, & \text{otherwise.} \end{cases}$$

where d is the distance between the *attribute* values of agents adjacent to a link, and v is the the maximum possible distance between the attributes of agents adjacent to a matching link.

Figure 5.15 shows some improvement in the clique covering for the trial with p_{end_3}, the end distribution now centers around 8 with a lowest value of 3. For a sample trial we find fewer, 21 out of 277 or 7.6%, cliques with an attribute value range that contains 30. Over all trials we find a lower average percentages of 7.1%, though this is partly due to the fact that more cliques are formed.

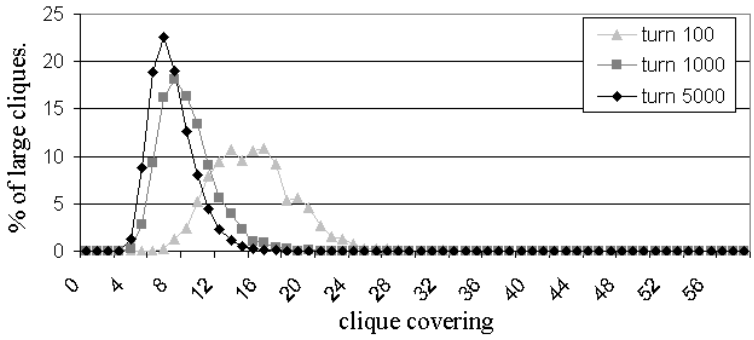


Figure 5.15: Clique covering distribution, agent-agent link lengths; p_{match_2}, $t = 0.5$, p_{end_3}, $v = 7.5$, p_{type_3}, $r = 6$.

5.3.2 Selectively Choosing Clique Connections

In general we find longer link lengths and a larger number of cliques per trial when the choice of matching links and choice of connections is no longer based on the same measure. Cliques, however, have an advantage over individual agents when upgrading and downgrading links since they have access to a list of all available link strengths. This means that cliques are able to actively choose their best links as connections, rather than having to make independent decisions about each link, as has be done up to this point.

Figure 5.16 shows the resulting clique covering when the connecting procedure is changed to successively upgrade the strongest matching links into connections, stopping once the maximum cluster size prevents the clique from joining with a neighboring clique. Note that this does not make all possible connections, as a strong connection that cannot be made to a large neighboring clique will prevent weaker connections to smaller cliques being formed. We found this behavior was preferable to simply making all connections that could be made, since it leaves room for the better connections to form once weaker connections are eventually downgraded.

```
process searchForConnections {
  whenever (``clock phase 1'' ){
    while (``first connection this turn'' | ``last connection successful'' ){
      bestMatch:Port := members.all.acquaintances.min( ``link distance'' , ( matching = TRUE & connecting = FALSE ) );
      if ( bestMatch.neighbor.myAgent.c.requestConnection( self ) = ACCEPT ){
        bestMatch.connecting := TRUE;
        ``join the cliques'';
      }
    }
  }
}
```
(5.1)

Figure 5.16 shows an improvement in the clique covering distribution, now centered on 7. For a sample trial we find 18 out of 492, or 3.7% of cliques with an attribute value range that includes 30. Thus again we find fewer cliques that could contain an agent with attribute 30, though more cliques in general. Over all trials we found minimum, average, and maximum values of 2.1%, 3.3% and 4.7% respectively, for this measure.

We attempted a number of changes to further improve clique strength which we will

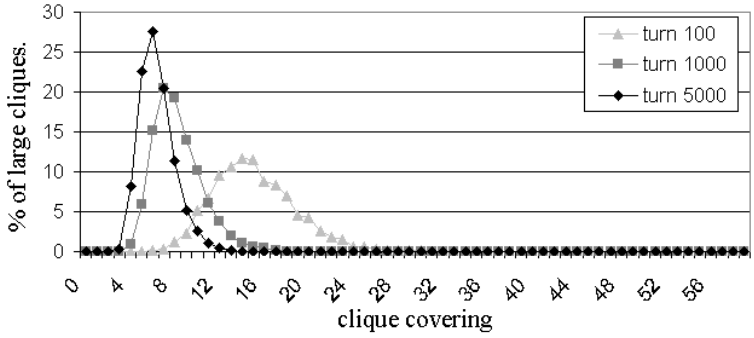


Figure 5.16: Clique covering distribution; only best connections made, p_{match_2}, $t = 0.5$, p_{end_3}, $v = 7.5$, p_{type_3}, $r = 6$.

only summarize. One successful method was to allow cliques to actively choose weaker connections to downgrade, to make room for stronger matches to be upgraded. Joining and dividing cliques, however, is an expensive operation. Therefore, we found that it is in fact best to modify clique operations to evolve cliques at a slower pace. Accordingly, we again changed clique behavior so that at each turn either the best matching link is upgraded to a connection, or, if this was not possible because of the size limit, the worst connection is downgraded, provided it is weaker than the best matching link.

We also found that with a more precise selection of connections, the matching links made played less of a role in cluster strength. We therefore increased the allowed distance between matching objectives from $t = 0.5$ to $t = 10$. Since this allows more matches to be found, it results in initial cliques forming more quickly, and in larger cliques developing. When we increased t further to $t = 30$, however, we found that large cliques form, but that the choice of matching links was not selective enough, resulting in a clique composition that remained ridged rather than improving over time.

Since cliques now intentionally downgrade connections we also considered removing the random ending of matching and connecting links all together. We found, however, that random ending still played a small role in improving clique strength, probably through removing unwanted links and thus increasing the rate at which good matching links are found. We thus experimented with a number of end probability functions to allow stronger links to be more highly favored, and found that the following worked well:

$$p_{end_4}(d) = \begin{cases} 0.01 * d/v + 0.00001, & \text{if } d < v \\ 0.01, & \text{otherwise,} \end{cases}$$

with $v = 3$. In fact we found that improving the end probability function worked so well that it had a far greater effect than the intentional breaking of links by cliques. Thus we again changed the clique behavior to attempt to make only the best matching link into a connection each turn, but to do nothing if this was not possible.

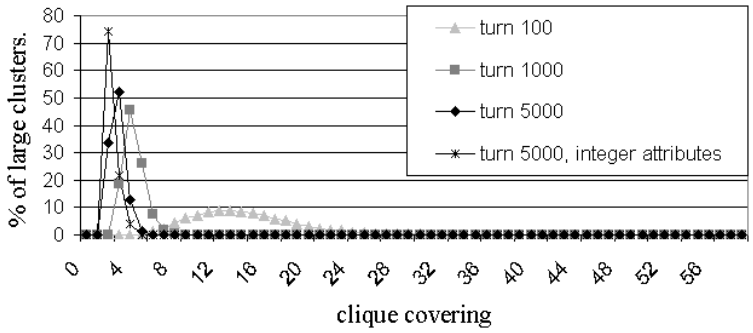


Figure 5.17: Clique covering distribution; only best connection made each turn, p_{match_2}, $t = 10$, p_{end_4}, $v = 3$, p_{type_3}, $r = 6$.

```
[Clique] process searchForConnections {
  whenever (``clock phase 1'' ){
    bestMatch:Port := members.all.acquaintances.max( ``link distance'' , ( matching = TRUE ) );
    if ( bestMatch.neighbor.myAgent.c.requestConnection( self ) = ACCEPT ){
      bestMatch.connecting := TRUE;
      ``join the cliques'';
    }
  }
}
```
(5.2)

Figure 5.17 shows the sum effect these changes have on clique strength. We see that clique covering is greatly improved, nearly to the optimal range of 2 in turn 5000. Figure 5.18 shows for a sample trial the 6 cliques, out of 279, with an attribute value range that crosses 30. We see in comparison with Figure 5.14 that clique attribute ranges have become much tighter. Over all trials we found a minimum of 0.72%, an average of 1.5% and a maximum of 2.6% of cliques included 30 in their attribute range. The data set "integer attributes" in Figure 5.17 shows the clique covering distribution at the end of trials if we use the same agent and clique behavior, but give agents integer attributes instead of real values. With integer attributes strong cliques become easier to find and we see that the clique covering distribution improves even further.

Ideally we would like cliques to completely divide the attribute space so that only one clique would include 30 in its attribute range. This is, however, difficult when attributes are picked from a continuous interval, as agents connected by a path that contains only short connecting links can still have attributes that are a large distance apart. To explore this further we give agents integer attributes, picked from the range $[1,M]$. For M=60 and N=5000 we expect to have about 83 of each agent type. Thus a cluster of size 30 is too small to include all of the agents of a particular type. We therefore experiment with larger cliques to see if we can group all of the agents of a single type together into one clique. We consider S = 80, 120, 160. Table 5.1 shows results for 24 trials run for each of these values of S.

We see in the Table 5.1 that all agents of one type do congregate in a single clique when the cliques are larger than the expected number of agents. When $S = 80$ cliques are slightly too small to fit all agents for all attribute types. Considering the cliques in turn 5000 of a

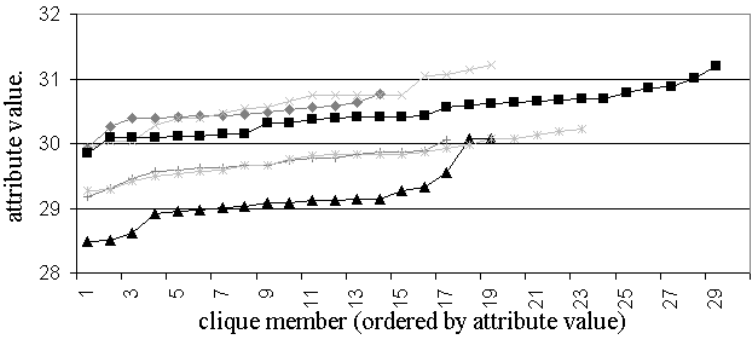


Figure 5.18: Cliques whose attribute range crosses 30 for a sample trial from Figure 5.17.

sample trial with $S = 80$, we find 8 cliques out of 102 had more than one type, of these one had 3 different types, the others 2 different types. We further found that a total of 23 agents were placed in a clique which was not the clique with the most agents of their attribute value. Of these agents that were out of place, all were in the clique with agents of either the type one above or below theirs. For a sample trial with $S = 120$, on the other hand, we found 60 cliques, one for each type of agent. Of these, 8 cliques had one agent from the type above or below. For a sample trial with $S = 160$ some adjacent types had few enough members to fit into one clique. We thus found 52 cliques in total with 11 containing agents of 2 different types. Of these, 2 had one agent from the type above or below, the rest contained all of the agents of their two different types. There was also one clique with a single agent which should have been in one of these latter cliques.

The fact that agents with integer attributes can separate themselves into homogenous cliques suggests that agent attributes should be chosen so as to create natural groups, rather than from a single continuous interval, if we desire to create good grouping behavior.

Table 5.2 shows results for trials in which the attributes of agents were chosen in such a way as to create gaps between groups of agents. For example, in line two attributes were chosen from the intervals $[x-0.25, x+0.25]$, around each integer x in the set $1,\ldots,M$. This gives clear groups of attributes with a gap of 0.5 between each group. We see that even a small gap allows groups to be more clearly identified by the agents, and when groups contain identical agents (gap=1) agents can be classified correctly.

5.4 Learning the Optimal Matching Range

The model developed so far in this chapter contains a large number of agent and clique parameters:

- δ, the number of objectives and links per agent. A large δ supports a much higher objective range, M, since agents each have a larger search space and thus the probability of any two randomly picked objectives matching can be lower. On the other hand, a larger δ costs agent resources, and δ is not necessarily easy to adjust since increasing δ involves somehow discovering a new neighbor.

	number of cliques containing attribute 30			fraction of cliques containing attribute 30		
S	min	avg	max	min	avg	max
S=30	3	4.62	9	0.0112	0.0178	0.0344
S=80	1	2	3	0.0100	0.0198	0.0312
S=120	1	1	1	0.0164	0.0166	0.0167
S=160	1	1	1	0.0192	0.0200	0.0213

Table 5.1: Clique containing attribute 30 for agents with integer attributes; 50 trials per data set.

- *S*, the maximum number of agents per clique. The best value of *S* depends on the size of the actual cliques in the underlying data (if they exist), or on the number and relevance of replies wanted for a query (if we think of the cliques as forming a directory structure).
- *r*, the size of the interval of objective types for an agent. The value of *r* depends on what agents actually represent. Multi-purpose agents, or agents with many abilities, have a large *r*, agents that are very specific or have limited abilities have a small *r*. It is unlikely that an agent's *r* can easily be changed.
- *t*, the maximum distance between matching objectives. The larger *t* is the more flexible agents are when accepting matches and the larger *M* can be. As with *r*, however, the value of *t* depends strongly on the application. *r* and *t* must be large enough that agents actually find matching links, but small enough that these matches are meaningful or useful.
- *v*, which determines how likely it is that a matching and connecting link of a particular length is kept. This value is mainly determined by *t* which determines the possible lengths of matching and connecting links. In the previous sections, *v* appeared to have the most effect on the strength of cliques. The more picky cliques are about which matching links and connections to keep, the higher the clique strength.

A major disadvantage at present is that these parameters are hand adjusted according to *M*, the size of the interval of attribute/objective types, to obtain a system where high quality cliques will form. *M*, however, is a global value and is thus unknown to individual agents. In this section we attempt to design agents that can learn the appropriate values for their parameters based on the quality of cliques they are forming (or not forming). We focus on agents that adjust to an unknown value of *M* by adjusting *v* and *t*. These parameters can be thought of as the most flexible since they are values that individual agents or cliques are most able to change. Adjusting *S* on the clique level should also be possible, but turns out to be much more difficult. In Chapter 8 we look further at allowing clique size, *S*, to change based on the quality of the cliques found.

Adjusting the maximum length of matching links

When *M* is unknown an agent does not know what strength of matches between objectives it can expect to be available. It thus has no basis for determining if a candidate nonmatching link is "good" or not. For our current agents this decision amounts to choosing *t*, the range

	number of cliques containing attribute 30			fraction of cliques containing attribute 30		
s	min	avg	max	min	avg	max
gap 1	1	1	1	0.0164	0.0166	0.0167
gap 0.5	1	1.2	3	0.0154	0.0196	0.0492
gap 0.1	1	1.6	3	0.0137	0.0241	0.0469
no gap	1	1.74	4	0.0135	0.0259	0.0625

Table 5.2: Clique containing attribute 30 for agents varying gaps between attribute sets; 50 trials per data set.

of the match probability function. If t is set too low agents will be searching for matches that are unlikely to exist and thus will never find them. On the other hand if t is set too high, agents will readily accept poor matches and thus not discover better matches that could probably be found. Ideally, t should be just large enough that only the best matches in the current search space are accepted as matching links. Agents can thus determine if their current value of t is appropriate by measuring how easily they discover matching links. If an agent cannot find a match within a reasonable amount of time it needs to increase t, while if it finds matches quickly it can decrease t. Any learning function should be usable to achieve this behavior. For now we simply pick one, analogous to that used in by the bidding agents in Chapter 4, to experiment with. We redesign agents to adjust t as follows.

- **Increasing t:** When agents cannot find a match we would like them to increase t slowly. A fixed rate of adjustment, however, is inappropriate since the total change in t that might be needed is unknown. Agents thus watch the matches offered to all of their objectives for some time period to allow them to estimate a good target value for a new t. In the following experiments we use a time a period in which an agent observes 100 candidate link strengths. Agents slowly increase t to a target distance of the lowest distance seen in that time period. This is achieved by increasing t by 1/100 of the distance between the original t and the target distance for 100 turns. An immediate jump is inappropriate since other agents are simultaneously adjusting. If a lower link distance is seen during this time period the target range is reset. Agents then return to the observation phase. If a distance is seen that is within the current range in either the observation or adjustment period the process is restarted since this indicates that the current range is appropriately large.

- **Decreasing t:** When a matching link is found, t is set to the length of that link. We keep the triangular shaped $p_{match_2}(d)$ function, as used in Section 5.3, since making closer matches more readily than further ones helps to lower t more quickly.

This learning procedure can be implemented within the Port *matches* function in the pseudo-code from Figure 5.1[2].

```
[Port] internal procedure matches( p2:Port ) returns ( Boolean ){
  ``update the matching range according to learning function'' ;
  return(``true with probability p_match(d)'';
  //Where d is the distance between objective and p2.objective.
}
```
(5.3)

[2]More detailed pseudo-code for learning expected matching link lengths is given in Section 8.3.

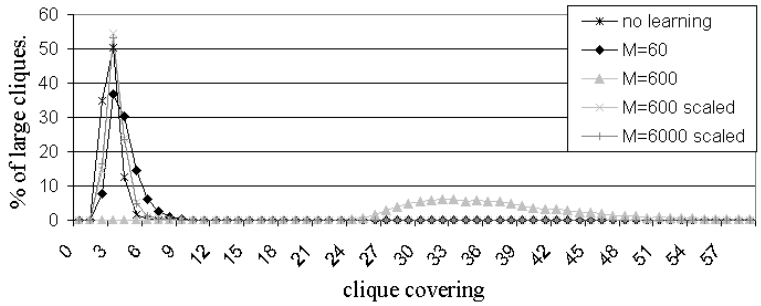


Figure 5.19: Clique covering distribution for agents which learn t.

Using the following parameters, we run 50 trials each using agents that learn as above and the old fixed t agents.

- N=5000, M=60, δ=5, S=30
- $p_{type_3}(d)$ with $r = 6$
- $p_{match_2}(d)$ with $t = 10$
- $p_{end_4}(d)$ with $v = 3$
- Each turn matching links are upgraded into connections from strongest to weakest, stopping after the strongest remaining matching link can no longer be upgraded. Cliques do not deliberately break connections.

To test the ability of the agents to adjust to M we also run 50 trials with M=600, v=30 and M=6000, v=300. As t changes v also needs to be adjusted, which, for this experiment, is done by hand. The next section discusses learning v as well.

Figure 5.19 shows the clique covering distribution of large cliques for agents that learn t compared to the previous agents which were given a fixed t. For M=60 the non-learning agents find slightly better cliques than the learning ones, thus knowing the correct M is an advantage, but not a large one. For larger M, cliques cover a greater distance since the attributes of the agents are further spread. But, if we scale the clique covering values for M=600 and M=6000 we find their covering distribution is as good the non-learning case. This indicates that agents are able to adjust to an unknown M by adjusting t.

Adjusting the end range, v

While the agents in the previous subsection are able to adjust t we still had to manually set v, the parameter that governs the rate at which matching and connecting links are downgraded, to fit M. Ideally, v also needs to be set based on the available link strengths. To do this we could use the range learned for t. On the other hand, cliques have a better source of information when deciding which matching and connecting links to downgrade: they each have a number of exiting matching and connecting links and can therefore set the probability

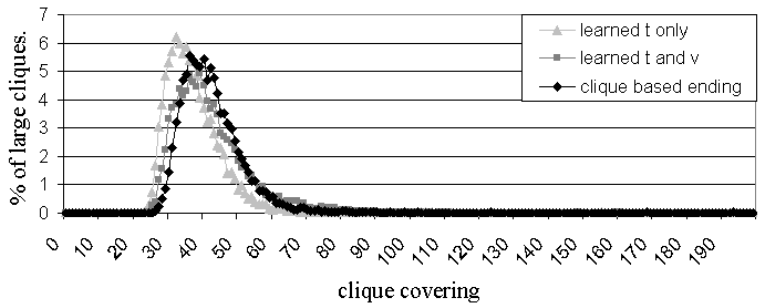


Figure 5.20: Clique covering distribution for agents which learn t and cliques which use relative downgrade probabilities.

of ending each link relative to the others. Accordingly, we make downgrading links a clique-level decision and choose links to downgrade based on the relative strength of other links in the clique.

We create a fixed probability of *one* matching or connecting link being downgraded in each clique each turn. This value is weighted by a clique's size so that large cliques, which have more matching and connecting links, are more likely to downgrade a link, giving small cliques a chance to grow. We set the probability of a clique, c, of size $|c|$ downgrading a matching link or connection each turn to $p_{end} = 0.003 * |c|$. Cliques determine which matching or connecting links to actually downgrade according to the following heuristic. Matching and connecting links are weighted by their distance, minus the length of the strongest link, and a random link is selected according to these weights. Thus given links of length 5, 3, 2, and 1 in a clique, their probabilities of being chosen to be downgraded are respectively 4/10, 2/10, 1/10 and 0. Notice that the strongest matching or connecting link never breaks. Because of this, we add the restriction that agents cannot form matching links between their own objectives. This functionality can be implemented within the Clique *reconsiderComposition* process.

```
[Clique] internal process reconsiderCompositon(){
  whenever (``clock phase 4'')){
    if ( ``if a objective should end according to p_end'' ){
      candidates: set of Port := ``all clique Ports for which matching = TRUE '' ;
      p:Port := ``choose a member of candidates, weighting by link distances'' ;
      p.reset();
      ``split off cliques if necessary'';
    }
  }
}
```
(5.4)

We run 50 trials for M=600 comparing the resulting agents with those that learned only t from the previous section, and agents that learn both t and v. Figure 5.20 plots clique covering data for this experiment. As with learning t, we see that clique-based link downgrading does not make up for the fact that M is unknown, but still produces reasonably strong cliques.

5.5 Conclusions

In this chapter we hypothesized that, for agents whose objectives are related to some central attribute, the matchmaking process could be used to form cliques of similar agents. In other words, ongoing peer-to-peer matchmaking could produce an overall system organization. We found that this could be done for agents with straightforward Real value attributes, but that it requires a more complex clique and agent behavior than is necessary for simple matchmaking. We first showed that a reasonable grouping behavior could be created by adjusting only the functions by which agents choose objectives, matching links, and connections. In order to obtain truly cohesive groups of agents, however, we needed to move the decision of what connections to form and downgrade up to the clique level, where each link could be compared with other clique links, rather than evaluated individually. We further showed that these agents could adjust to the scale of any data set, i.e. M, by adding a simple learning behavior. We, however, could not get cliques to perfectly separate a data set unless we introduced an underlying partitioning in the data itself. Moreover, we did not consider adjusting clique size, S to fit the data set. This last problem is considered in Chapter 8.

Chapter 6

Clustering 2D Spatial Data

By organizing into groups of similar items, the agents in the previous chapter are in effect solving a clustering problem. In this chapter we give the agents developed in Chapter 5 two dimensional spatial data as attributes, and examine their ability to find straightforward clusterings of data. This work was originally presented in [26] and [28].

6.1 Introduction and Related Work

Clustering has been studied in a variety of fields, notably statistics, pattern recognition and data mining. These fields have a wide range of purposes in mind: discovering trends, segmenting images, or grouping documents by subject. Nonetheless, in all of these disciplines the underlying problem is the same: given a number of data items, group them such that items in the same group are more similar to each other than they are to items in other groups [14].

The creation of groups of agents, based on like interests, provides a potential alternative structure to enable search within very large decentralized systems, where maintaining a directory becomes too costly. Such groups place potential partners for collaboration in an agent's immediate local environment [11]. When interests of agents include jointly working on common tasks, this process evolves into coalition formation: the negotiation of agreements between agents with complementary skills for the distribution of work and rewards [21] [41] [42]. Clustering, as studied in this chapter, is a more basic problem, yet an essential part of the process of forming coalitions. A multi-agent system, made up of heterogeneous agents, that cannot somehow group similar agents is unable to introduce potential coalition members to each other.

Traditional clustering algorithms, however, cannot be directly applied in the decentralized setting we study in this work. Most algorithms for clustering focus on how to form groups given a file or database containing the items. Yet, for Internet applications like finding similar web pages or finding agents with similar interests, items can be widely distributed over many machines and the issue of collecting the items in the first place gains importance. Centralized clustering is problematical if data is widely distributed, data sets are volatile, or data items cannot be compactly represented. Decentralization, on the other hand, is a thorny problem. Even in the centralized case where each data item can be compared to every other data item, perfect clusters can be hard to find. Decentralization creates the additional complication that even if a correct classification can be determined with the

incomplete information available, the location of items belonging to each class also needs to be discovered.

In general, studies of clustering assume that data contains natural groups that need to be discovered. The data sets studied in Chapter 5 contained random points, and therefore there were no obvious boundaries between the groups of agents formed. In actual application data, however, it is fairly reasonable to assume that data will separate more obviously into particular categories. There are a large number of algorithms for discovering these natural clusters, if they exist (see [15] for a general review of the literature). The majority of these algorithms focus on finding clusters given various properties of the data set: clusters of widely differing sizes, odd cluster shapes, little separation between clusters, noise, outliers, high-dimensional data, and complex data types for which a similarity function is difficult to define. In general, such clustering algorithms focus on creating good compact representations of clusters and appropriate distance functions between data points. To achieve this purpose they generally need to be given one or two parameters by a user that indicate the types of clusters expected. Most commonly, algorithms are given the number of clusters into which the data set is to be split or a density value that defines the expected distance between points within clusters. Since a central representation is used where each point can be compared to each other point or cluster, points are never placed in a cluster with hugely differing members. Mistakes made by these algorithms instead take the form of incorrectly splitting a real data cluster in half or incorrectly combining two neighboring data clusters into a single cluster. It is not, however, always obvious where such cluster boundaries should be placed. As a whole, the definition of clustering is imprecise: the creation of clusters in which points have more in common with other cluster members than with members of other clusters. Given the complexities listed above it is usually not entirely clear what the "correct" clustering of a data set is. Therefore, there is generally no one best algorithm for obtaining good clusters [14]. The appropriate algorithm depends on the peculiarities of the data set considered and the intended use of the clustering produced.

This chapter focuses on yet another complexity that must be faced when studying clustering in a decentralized agent setting: the distribution of the data over many machines. Basic data sets in which clusters are clearly separable, two-dimensional, of equal size, and circular in shape are considered. As these data sets do not contain any of the difficulties addressed by more complex algorithms, the clusters found by our multi-agent system are compared to those found by Forgy k-means clustering as described in [14]. This is the simplest of the centralized clustering techniques. Nonetheless, it works well on the elementary data sets examined and illustrates the basic abilities and common mistakes of centralized clustering.

An orthogonal line of research to the quality of clustering is the speed of clustering algorithms. The amount of time it takes to find a clustering is important for large data sets. K-means clustering has the lowest time complexity of the standard centralized clustering algorithms. It chooses a set of k cluster centers, called centroids, and then compares each of the n points in the data set to each centroid, placing them in the cluster to which they are closest. It recalculates centroids based on the resulting cluster memberships and repeats this process, called a k-means pass, l times. Its time complexity is thus $O(nkl)$, however, k and l are usually small compared to n and thus it is usually considered to be $O(n)$. K-means is a partitional algorithm, meaning that it creates a single partition of the data. More

complex, and thus more accurate, algorithms are generally hierarchical. They build a table of the distances between every pair of points in the data set and use this table to create a series of partitions, each time splitting or combing clusters based on the next largest or smallest distance between points. However, the construction of the distance table has time complexity $O(n^2)$, making hierarchical algorithms unsuitable for large data sets.

A number of papers have addressed techniques for dealing with large data sets. One way of reducing the execution time is to reduce the space of solutions that a clustering algorithm has to search. CLARANS [25] is a k-medoid based algorithm that uses randomized search. K-medoid algorithms are similar to k-means algorithms except that they represent clusters by their medoids, the most central member of the cluster, instead of their centroids, the mean value point of the cluster. Thus for a given k it is possible to search all possible sets of k medoids. There are, however, $\binom{n}{k}$ of these. CLARANS reduces this search space by using a randomized hill climbing technique. Starting with a set of medoids it checks at random only some of the sets that differ by one medoid for an improvement in the clustering. This random search, when restarted a few times in different places is shown to still produce good clusters while increasing the efficiency over exhaustive search. An alternative way of reducing the problem space is to consider only a sampling of the data points. CLARA [20] does this before running a k-medoid algorithm. CURE [13] also uses random sampling, combined with other techniques, to extend the range of a hierarchical clustering algorithm. Sampling methods assumes that a large enough sample contains a sufficient number of points from each cluster and thus needs to be given a known minimum cluster size. BIRCH [55] also does hierarchical clustering on a reduced sample space. BIRCH, however, uses a pre-clustering phase in which it first creates a compact summary of data points, called a CF-tree. Nodes of the CF-tree represent high density groups of points that are defined to be in the same cluster based on some threshold distance. CF-tree nodes are then sparse enough to be clustered using a hierarchical algorithm.

A second method of dealing with large data sets is to partition them into a series of smaller problems. CURE, in addition to sampling data, divides the resulting sample into partitions and separately pre-clusters each one. The resulting sub-clusters are then combined to create a final clustering. P-CLUSTER [19] parallelizes k-means clustering by partitioning the data set over a network of workstations. A server workstation determines and distributes an initial set of centroids and combines the information on the resulting clusters in each partition. It then distributes new centroid information for the following k-means pass. Other methods of parallelizing hierarchical clustering by distributing the calculation and storage of the distance matrix are surveyed in [34]. Density-based algorithms create very fine partitions of data by placing two points in the same cluster should they be close enough together and in an area with sufficiently many other points. This requires creating a data structure that can efficiently find the nearest spatial neighbors of a point. DBSCAN [8] does this by defining clusters as areas with a threshold density of points within a given radius and growing clusters out from single points based on this density.

All of the above methods can be used to find clusters of similar agents in multi-agent systems. Methods based on partitioning that limit, or eliminate, the global data that needs to be maintained are nevertheless best suited for very large-scale systems. The agent method we study in this chapter is based on the concept of creating a fine partitioning of points. Points are agents, giving us a high level of local decision-making capability. Two agents

are defined as belonging to the same cluster if they are sufficiently close together. However, unlike density-based methods, agents do not have a view of all of their spatial neighbors. Instead, agents choose cluster partners based on the closest agents they encounter and later abandon more distance partners in favor of new found closer ones. Defining when partners should be dropped can either be based on a minimum distance as in density-based algorithms, or a fixed maximum cluster size corresponding to the k used in k-means clustering. In this chapter we take the defined maximum cluster size approach. In Chapter 8 we study a more complex density-based approach. Although all data points (or agents) are considered in our analysis of the clusters, individual agents see only a sample of the data points at any one time. Agents work in parallel. Efficient use of resources is not a goal, instead we rely on fine grain parallelization and partitioning to allow us to handle very large data sets.

6.2 Model

In this section we adapt our agent model to fit the clustering problem, and define the method by which we will measure the quality of the clusters that agents produce.

6.2.1 Clustering Agents

In this chapter we consider the cliques found by our agents to comprise a clustering of the agent data set, X, represented by the attributes of agents. Each agent clique thus corresponds to a data cluster. The agents used in this chapter are basically the grouping agents from Chapter 5, with two-dimensional points as objectives, in place of tasks. We accordingly use the Euclidean distance between agent attributes to define the length of links. Figure 6.1 gives the pseudo-code for the resulting agents. As can be seen in the pseudo-code there are also a few further small modifications.

Instead of each port having its own objective and learning its own matching range, ports directly use their agent's attribute as their objective (line 8), and the matching range is learned by the agent (line 25) according to the learning procedure described in Section 5.4. In effect this means that $p_{type}(d)$ is 1 for distance 0, and 0 otherwise. This change simplifies the agents, which now rely purely on learning the matching range to achieve the flexibility needed to find initial matches.

We also slightly modify the manner in which connections are chosen to allow us to tune the speed at which cliques form (lines 39-44). During each turn, instead of making only the best possible connection, we recursively upgrade the best matching link until an upgraded fails because of the clique size restriction. We choose matches and connections to end according to the weighting function described in Section 5.4 (line 56). We use $p_{end} = \lambda|c|/S$, where λ is a speed parameter, $|c|$ is the number of agents in the clique, and S is the maximum clique size.

6.2.2 Measuring Cluster Quality

The experiments in this chapter cluster data sets of varying numbers of points. These data sets are generated according to the procedure described in [55]. Each data set consists of k clusters of 2-dimensional data points. A cluster is characterized by the number of points per cluster ($n_{\text{low}} = n_{\text{high}} = 100$) and the cluster radius ($r_{\text{low}} = r_{\text{high}} = \sqrt{2}$). The *grid* pattern is used, which places the cluster centers on a $\sqrt{k} \times \sqrt{k}$ grid. The distance between the clusters

```
1.  object Port{
2.    neighbor:Port; matching:Boolean; connecting:Boolean; myAgent:Agent;
3.    internal procedure reset(){
4.      connecting := FALSE; matching := FALSE;
5.      neighbor.reset();
6.    }
7.    external procedure linkDistance(){
8.      ``Euclidean distance between myAgent.Point.value and neighbor.myAgent.Point.value''
9.    }
10. }
11. object Point{
12.   value: [Real, Real];
13. }
14. object Agent{
15.   acquaintances: internal set of Port; //δ links per agent
16.   c: Clique; //The clique of which this agent is a member
17.   attribute: internal Point;
18.   matchingRange: Real;
19.   process searchForMatches [for each aq in acquaintances]{
20.     whenever ( ``clock phase 3'' & aq.matching = FALSE & aq.connecting = FALSE ){
21.       aq.neighbor := c.tradeIn( aq.neighbor );
22.       aq.matching := agentsMatch( aq, aq.neighbor )
23.   }
24.   internal procedure agentsMatch( p1:Port, p2:Port) returns ( Boolean ){
25.     ``update the matching range according to learning function'' ;
26.     return ( ``true if tasks match according to p_match'' );
27.   }
28. }
29. virtual object Clique {
30.   members: set of Agent; //All agents that are members of this clique
31.   unwantedAcquaintances: set of Port;
32.   external procedure tradeIn( p:Port ) returns ( p':Port ){
33.     unwantedAcquaintances.add( p );
34.     wait until( ``clock phase 2'' );
35.     return unwantedAcquaintances.remove( ``random element'' );
36.   }
37.   process searchForConnections {
38.     whenever (``clock phase 1'' ){
39.       while (``first connection this turn'' | ``last connection successful'' ){
40.         bestMatch:Port := members.all.acquaintances.min( linkDistance(), ( matching = TRUE & connecting = FALSE ) );
41.         if ( bestMatch.neighbor.myAgent.c.requestConnection( self ) = ACCEPT ){
42.           bestMatch.connecting := TRUE;
43.           ``join the cliques'';
44.         }
45.       }
46.     }
47.   }
48.   external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when (``available''){
49.     if (members.size + requester.members.size ≤ S ) return ACCEPT;
50.     else return REFUSE;
51.   }
52.   internal process reconsiderCompositon(){
53.     whenever (``clock phase 4''){
54.       if ( ``if a task should end according to p_end'' ){
55.         candidates: set of Port := ``all clique Ports for which matching = TRUE '' ;
56.         p:Port := ``choose an element of candidates, weighting by link distances'' ;
57.         p.reset();
58.         ``split off new clique if necessary'';
59.       }
60.     }
61.   }
62. }
```

Figure 6.1: Psuedo code for 2D clustering version of the agents.

is controlled by k_g, which is set to 8. The noise parameter is set to 0, so that no random noise points are added to the data set. This creates a grid of well-separated, circular, 2D clusters with 100 points each and equal density.

Four data sets are generated with 25, 100, 400, and 1600 clusters labelled 5×5, 10×10, 20×20, and 40×40, respectively. A corner of the 20×20 data set is shown in Fig.6.2. We compare the quality of clusters found by our method to the generated clusters (perfect case), a set of clusters of the correct size but with randomly assigned points (random case) and the clusters generated by the Forgy k-means algorithm as described in [14] given the correct value of k, initial centers picked uniformly at random from all the points, and run until no further improvement in clustering is found.

In this thesis our main interest is in the groupings formed by the agents, as represented by clique membership, not the relationship between individual agents, as represented by links. We thus use measures from the the traditional clustering literature to assess cluster quality. These metrics consider the appropriateness of the boundaries placed between agent groups. Several measures of cluster quality are compared. First, the total squared error metric, E^2, which is used by the k-means algorithm to measure the improvement in its successive clusterings. Given k clusters $C_1, \ldots, C_k$, where C_i has a mean value m_i for $1 \leq i \leq k$,

$$E^2 = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - m_i\|^2.$$

Total squared error gives an easily computed measure of the compactness of clusters, but in doing so favors small clusters. In fact, clusters with a single point have a total error of zero and thus the total squared error alone cannot be used to compare clusterings of a data set with different numbers of clusters. The more points there are in a data set and the larger the range over which the data set is spread, the larger the total squared error. Alternatively, the *weighted average cluster diameter* used by Zhang *et al.* [55], gives another measure of the compactness of clusters. The *average pairwise distance* for a cluster C_i with points $P = \{x_1, \ldots, x_n\}$ is given by

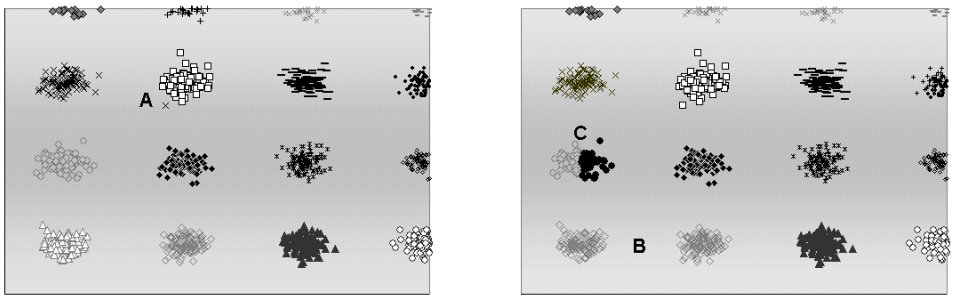
$$D_i = \frac{\sum_{p=1}^{n} \sum_{q=p}^{n} d(x_p, x_q)}{n(n-1)/2},$$

where $d(x_p, x_q)$ is the Euclidean distance between x_p and x_q. The weighted average cluster diameter for k clusters is then given by:

$$\overline{D} = \frac{\sum_{i=1}^{k} n_i(n_i - 1)D_i}{\sum_{i=1}^{k} n_i(n_i - 1)}.$$

While this measure scales with the number of data points and the number of clusters it also favors smaller clusters and does not account for singleton clusters.

Jain and Dubbes [14] describe several measures for comparing two clusterings of the same data by creating a contingency matrix which lists the number of points in common between each pair of clusters, between the two clusterings. These measures consider cluster membership, rather than the distance over which clusters are spread. The Rand statistic, described in [14], sums the number of pairs of points that are correctly placed in the same



(a) Clusters produced by the agent-based procedure. (b) Clusters produced by the k-means algorithm.

Figure 6.2: Example agent-based and k-means clustering results.

cluster and the number of pairs of points that are correctly placed in different clusters and normalizes by the total number of possible pairs. For our data sets which have many small clusters, however, the number of pairs correctly placed in different clusters dominates. Thus we also use the contingency matrix to calculate the number of points incorrectly placed by associating with each real cluster the found cluster with which it has the most points in common. We sum the number of common points over all real clusters and subtract from the total number of points to get the points that are out of place. This gives a clearer distinction between clusterings that are close to, but not quite, correct. On the other hand it does not distinguish clusters containing points that are incorrectly grouped into a single cluster. It also can count up to half of the points in a real cluster as incorrectly placed if that cluster is simply split in two.

6.3 Types of Clusters Found

Figure 6.2(a) shows the clusters found by our agent procedure in a sample trial for a section of the 10×10 cluster grid. Figure 6.2(b) shows, for comparison, the clusters found by an example k-means trial. Both methods were given the correct input parameters. The agent-based procedure used a maximum clique size, S, of 150 and $\lambda = 0.3$ The k-means algorithm was run with $k = 100$. Overall both methods found the correct number of clusters, meaning that k-means did not loose any, which is possible, and that cliques adjusted to the correct cluster size of 100 instead of staying at the maximum size of 150. Total squared error for the k-means trial was 126,889 versus 20,372 for the agent-based method.

The graphs in Figure 6.2 show the typical mistakes made by each method. At point A the agent method has associated a point with a neighboring cluster instead of with the correct cluster. This can happen when an agent has joined the correct clique, but has been broken off again due to the randomness of the function that chooses connections to downgrade. Generally, these agents reattach, though they can spend some time as a member of a neighboring clique. Thus at any point in time after cliques have found a good clustering, several such mistakes are likely to exist. Another way that this can happen is if an agent simply never finds the correct clique. This results in cliques with one or a small number of agents that can take a long time to find their correct group. This problem is related to the speed factor, λ, in p_{end}, which we discuss further in Section 6.4. The k-means algorithm,

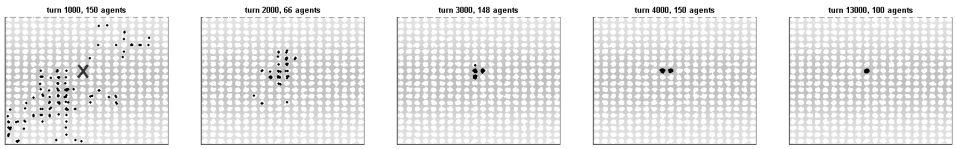


Figure 6.3: Cliques containing the point X (shown in turn 1000) over time.

by contrast, makes very different types of mistakes. At label B it has incorrectly joined two clusters into one, and at label C it has incorrectly split a cluster in two. This can occur when two initial centroids are chosen from the same cluster. This cluster then becomes split, but somewhere else two clusters need to be joined to maintain k. When there are large numbers of clusters it is likely that two initial centroids will be chosen from the same cluster. There are heuristic ways of choosing better initial centroids, but a perfect choice would amount to knowing the correct clustering a-priori.

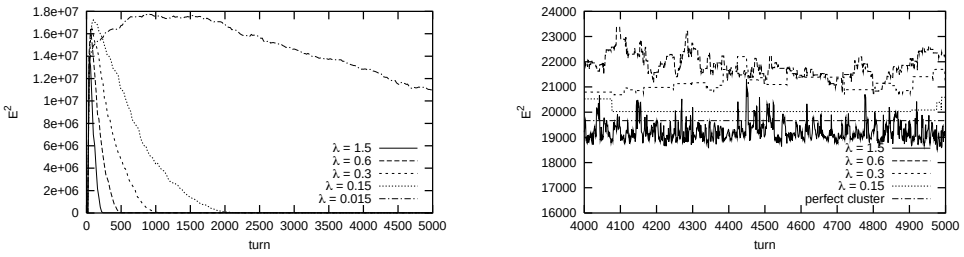
Figure 6.3 shows how the agent clique containing the point X develops over time for the 20×20 cluster data set. Notice how the clique is initially wide spread and contracts over time. Also notice that the clique when expanded stays near its maximum size of 150 agents but that once it has contracted it adjusts to the correct cluster size of 100 agents. In further experiments we found that cliques adjust to the largest real cluster size under their maximum size. Thus when S=199 points, agents can correctly find clusters of 100 points but a size limit of 200 points results in the combination of neighboring real clusters to create 200 point cliques. It does help, however, for cliques to have a little extra breathing room: $S = 150$ instead of 100 allows small cliques to move easily by temporarily joining larger ones, and thus helps them to find their correct clique should they become stranded.

6.4 Rate of Finding Clusters

Figure 6.4 shows the total squared error as a function of time for several example runs on the 10×10 data set. Time is measured in turns as defined in Section 2.5.1. The total squared error, E^2, begins high when cliques are spread out, (though it is initially 0 as all cliques begin at size 1) but rapidly converges to near its optimal value. This convergence is followed by a tail behavior where cliques improve slowly until an equilibrium is reached. Note that E^2 does not precisely correspond to the quality of clusters. A better clustering can have a higher E^2 than a worse one. Closer inspection through further experiments shows that during the tail there is a period where E^2 remains approximately constant while the clustering is actually improving slowly.

Figure 6.4 depicts the convergence for several values of the speed parameter, λ, used by p_{end}. Figure 6.4(a) shows that increasing λ increases the rate of convergence. Figure 6.4(b), shows, however, that λ also effects the stability of the equilibrium found in the tail. A slower speed results in more stability. $\lambda = 1.5$ results in the lowest E^2 value because many small cliques are created. The trial with the most stable equilibrium, $\lambda = 0.15$, is also the one with the best clustering. This can be seen by comparing the distribution of clique sizes for $\lambda = 1.5$ and $\lambda = 0.15$ in Figure 6.5.

From Figure 6.5, which shows the distribution of clique sizes at turn 5000 for two sample



(a) Total squared error over time. (b) Tail of Fig6.4(a).

Figure 6.4: Total squared error over time for different speeds (λ).

trials, it is apparent that the solution found when $\lambda = 1.5$ has a large number of very small cliques. This occurs when initial cliques contract too quickly. It is then possible for a small set of agents to become isolated. Since this set of agents is quickly broken off of any neighboring clique it joins, it never gets the benefit of the wider view that a large clique possesses. Thus, it can take a very long time to find its correct clique. For the remainder of our experiments we use $\lambda = 0.3$ as it represents a good balance between rapid convergence and quality of the end solution.

6.5 Increasing System Size

Figure 6.6 shows how the speed of convergence changes as system size increases. In this figure we plot, per time, total squared error normalized by the number of data points, N, and the area, A, over which those data points are spread. Each curve represents the average values of 50 trials. The different test data sets are generated with 5×5, 10×10, 20×20, and 40×40 clusters, and thus with 2,500, 10,000, 40,000 and 160,000 points. We consider the smallest data set, 5×5, to have an area of 1. The larger data sets, 10×10, 20×20, and 40×40, thus have an $A = 4$, 16, and 32 respectively. The number of points in a cluster is kept constant when increasing system size. Preliminary experimentation suggests that agent-based clustering of systems with the same number of points but different numbers of clusters, possibly produces different behavior. A few large clusters are much easier to find than a large number of small clusters.

Figure 6.6 shows that while larger systems continue to converge rapidly, their improvement phase becomes more drawn out. Figure 6.7 shows the tail behavior of the data sets in Figure 6.6. Here E^2 is normalized only by N, since in all of the data sets real clusters are equally far apart. Hence the average total squared error, $\langle E^2 \rangle = E^2/N$, for the perfect clusters is about the same for all of the data sets. In Figure 6.7, the average total squared error, $\langle E^2 \rangle$, reaches its equilibrium value for the 2,500, 10,000, and 40,000 agent data sets, while for the 160,000 agent data set $\langle E^2 \rangle$ is still slowly improving. There is a small increase in the equilibrium value of $\langle E^2 \rangle$ for larger data sets.

Figure 6.8 plots the time it takes to reach certain values of $\langle E^2 \rangle$, again averaging $\langle E^2 \rangle$ over 50 trials for each point. This shows more clearly that better quality solutions take increasingly more time to find as the number of agents increases. Trials with many agents converge to higher final $\langle E^2 \rangle$ values, thus the two curves for the time to reach the lowest $\langle E^2 \rangle$

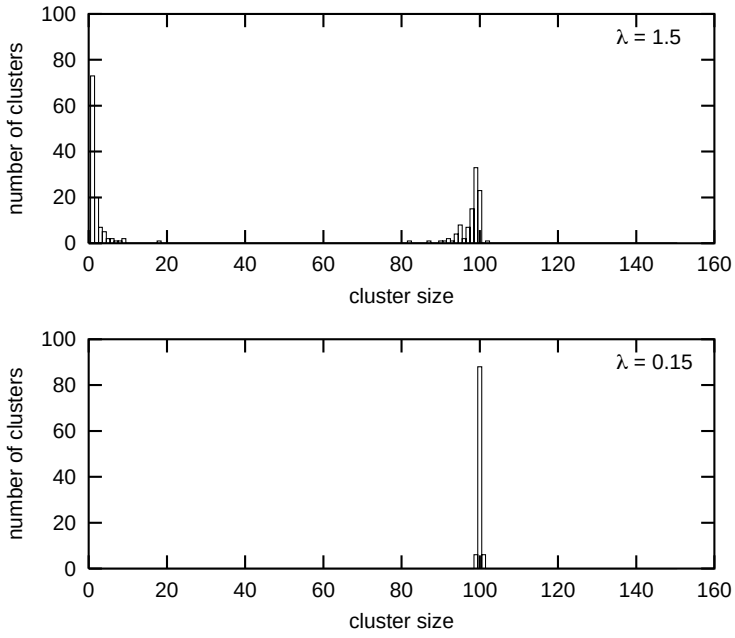


Figure 6.5: Distribution of clique (i.e. cluster) sizes: $\lambda = 1.5$ and $\lambda = 0.15$.

values are incomplete. Trails with very large numbers of agents never reach these values. This behavior can also be used to explain the fact that the last points in the curves that plot the time to reach the lowest three $\langle E^2 \rangle$ values are slightly higher than would be expected according to the times measured for trials with smaller numbers of agents. In, for instance, the $N = 160,000$ trials, the time to reach a $\langle E^2 \rangle$ of 10 is higher than expected because the $\langle E^2 \rangle$ = 10 is only slightly lower than the final $\langle E^2 \rangle$ value that can be found with this number of agents. This can be seen by examining the the behavior for 160,000 agents (the 40×40 curve) in Figure 6.7. In general, however, the cost of reaching a particular solution quality grows less than linearly with system size. This is an improvement on the linear costs of the *k*-means algorithm. Table 6.1 lists all the statistics for the different system sizes, run for 5000 turns with 50 trials for each data set, compared to the *k*-means algorithm run 100 times for each data set, the values for perfect clusterings, and the values for random clusterings. While the agent cliques usually do not find the perfect clusterings, they consistently improve upon the clusters found by the *k*-means algorithm. For smaller systems the agents place more than 99% of the points correctly and for the larger system, which was still improving, they place at least 95% of points correctly.

6.6 Conclusions

In this chapter we showed that the agent grouping behavior we explored in Chapter 5 can be extended for use with more complex data, creating a decentralized clustering algorithm. We compared the clusterings found by this decentralized algorithm to those found by the

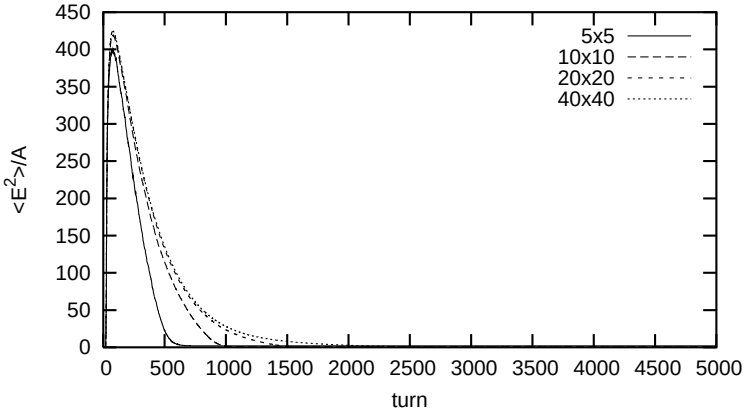


Figure 6.6: $\langle E^2 \rangle / A$ for different system sizes.

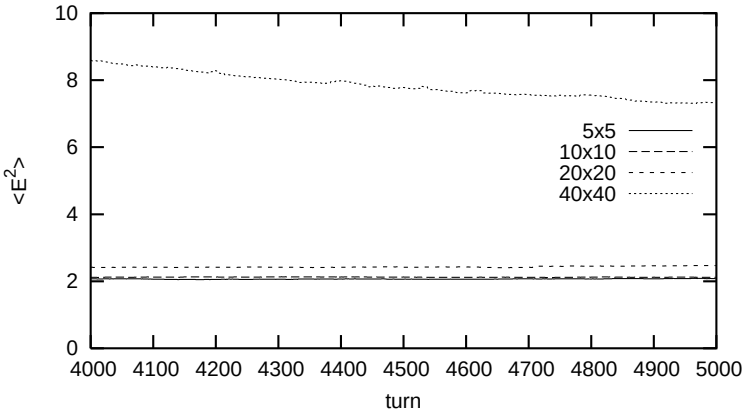


Figure 6.7: $\langle E^2 \rangle$ for the tail of Figure 6.6.

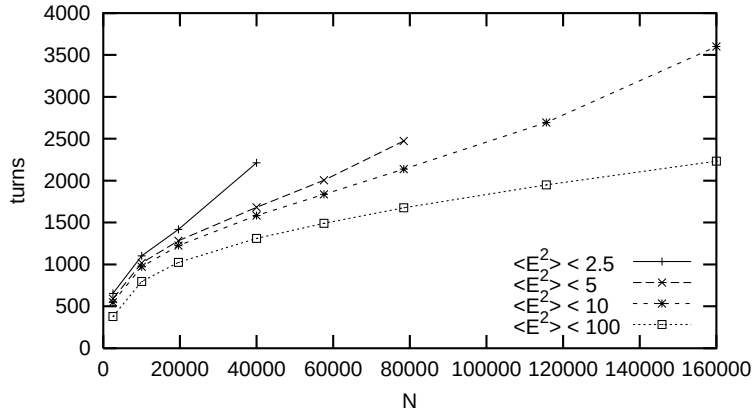


Figure 6.8: Time to reach various $\langle E^2 \rangle$ values with increasing systems size N.

	number of clusters			E^2			$\langle E^2 \rangle$	$\overline{D}$			Rand	points out of place			
number of points (agents)	min	avg	max	min	avg	max	avg	min	avg	max	avg	min	avg	max	avg % total
PERFECT CLUSTERINGS															
2,500		25			5044.03		2.02		1.79		1		0		0
10,000		100			19662.99		1.97		1.77		1		0		0
40,000		400			79350.99		1.98		1.77		1		0		0
160,000		1600			315993.13		1.97		1.77		1		0		0
RANDOM CLUSTERINGS															
2,500	25	25	25	1.27E+06	1.27E+06	1.28E+06	509.64	28.86	28.96	29.06	0.92354	2272	2289.61	2304	91.58
10,000	100	100	100	2.09E+07	2.09E+07	2.09E+07	2090.38	58.58	58.69	58.78	0.98030	9560	9576.37	9590	95.76
40,000	400	400	400	3.37E+08	3.37E+08	3.38E+08	8429.88	117.76	117.86	117.97	0.99504	38920	38943.87	38978	97.36
160,000	1600	1600	1600	5.40E+09	5.40E+09	5.41E+09	33773.95	235.79	235.90	236.01	0.99876	156744	156772.57	156811	97.98
KMEANS: k=100															
2,500	23	24.87	25	17466	29131	43853	**11.65**	3.14	4.13	5.75	0.98508	57	146.30	235	**5.85**
10,000	97	99.22	100	88575	119874	162271	**11.99**	3.55	4.20	5.13	0.99617	368	576.88	857	**5.77**
40,000	392	396.78	400	412301	485795	564576	**12.14**	3.87	4.25	4.66	0.99903	1939	2327.25	2730	**5.82**
160,000	1580	1588.54	1598	1778913	1923502	2083301	**12.02**	4.02	4.22	4.43	0.99976	31369	34398.64	37667	**21.50**
AGENTS: $L = 150, \lambda = 0.3$															
2,500	25	25.04	26	5044	5205	5553	**2.08**	1.79	1.80	1.82	0.99987	0	2.02	7	**0.08**
10,000	100	100.02	101	20215	21204	23031	**2.12**	1.78	1.79	1.81	0.99995	6	12.42	24	**0.12**
40,000	399	400.34	407	87979	98733	172886	**2.47**	1.80	1.83	1.95	0.99998	46	104.13	216	**0.26**
160,000	1610	1625.61	1647	1057818	1172274	1368215.61	**7.33**	2.59	2.70	2.82	0.99990	6320	7123.93	7996	**4.45**

Table 6.1: Experimental data summary.

k-means algorithm, a traditional centralized manner of clustering. We found that the quality of clusterings produced by the agents compared favorably, and that the scalability of the decentralized algorithm was better than that of the centralized version.

We did not, however, solve the problem of how to determine how many clusters are in a data set. Agent clusters still need to be given a fixed maximum size a-priori. This limit should pose no problems for applications where approximate cluster size is known, for instance the number of replies to a query, but does limit the applicability of agent clustering. While k-means also faces this difficulty, it is more essential to remove it in the decentralized case since it is less easy for an outside observer to study the end clusterings and adjust S for the agents, than it is for a user to adjust k for k-means. We study this problem further in Chapter 8.

The decentralized clustering approach is also limited by the fact that it requires a metric to measure the distance between data points, assuming that agents can assign comparable strengths to links. In this chapter we used a single absolute metric so that it was clear which links were stronger than others. In an application where agents are more independent and data is more complex, agents may each apply different metrics to measure link strength. It would thus be interesting to know how similar these separate metrics must be in order to achieve a reasonable clustering. Chapter 7 provides some insight into this question.

Chapter 7

Clustering Text

Studying the clustering of 2D spatial data provides a setting in which the quality of clustering algorithms can be easily measured. Real data, on the other hand, poses a far more complex problem. One of the reasons for this is the high dimensionality of most real data types, which makes suitable measures of point similarity harder to develop. In this chapter we demonstrate how the decentralized agent clustering algorithm considered in Chapter 6 can also be applied in a more realistic high-dimensional text-clustering setting. The work in this chapter was originally presented in [28]

7.1 Introduction and Related Work

Text clustering uses the same basic methods as spatial data clustering. It faces, however, the additional difficulty that the similarity of text documents is ambiguous. Since there is no strict distance measurement that can be used, text algorithms focus on effective ways to extract a document's meaning through identifying content bearing words in the text, or by means of any additional information available through user feedback or references [9]. Again, in this work we wish to keep the issue of decentralization separate from that of difficult to find clusters. Thus we consider a straightforward data set and use comparison methods from the classical word vector model [40].

The word vector model represents documents as vectors of the unique words they contain. Obviously, some words within a document convey more meaning than others. The word "the" for instance in this chapter tells us nothing about the chapter's overall subject, while the word "clustering" would be a useful keyword for indexing. A process called stopping can be used to remove the most used common words. Also, some words like "clustering" and "cluster" actually have the same meaning and thus need not be counted separately. Again, a process called stemming can be used to list only the unique stems of words in a document. On the other hand, as it is not certain that stopping and stemming are actually needed, we do not use them in our experiments.

More importantly, the word vector model recognizes that the frequency with which words are used gives an indication of their relative contribution to a text's meaning. It notes that very frequently used words, and very infrequently used words tend to convey less meaning. Moreover, for clustering, what actually distinguishes documents within a data set is the difference in the relative frequencies with which they use words. Thus the word "agent" might be used to classify this work, but would not be useful in grouping its

```
1.  object Point{
2.    text: ``full document text'' ;
3.    wordVector: set of Word; //vector of unique words in the text
4.    weights: set of Real;//weight for each word
5.    counts: set of Integer; //count of the time each word was seen in a neighbor
6.   external procedure updateWeights( p2: Point ){
7.     ``check if each word occurs in p2.wordVector, updating counts '' ;
8.     ``reassign word weights according to counts for each word '' ;
9.   }
10.   external procedure distanceTo( p2: Point ) returns ( Real ){
11.     ``distance to p2 according to some similarity coefficient'' ;
12.   }
13. }
14. object Agent{
15.   acquaintances: internal set of Port; //δ links per agent
16.   c: Clique; //The clique of which this agent is a member
17.   attribute: internal Point;
18.   process searchForMatches [for each aq in acquaintances]{
19.     whenever ( aq.matching = FALSE ){
20.       aq.neighbor := c.tradeIn( aq.neighbor );
21.       aq.matching := agentsMatch( aq.neighbor, myAgent )
22.   }
23.   internal procedure agentsMatch( a2:Agent ) returns ( Boolean ){
24.     attribute.updateWeights( a2.attribute );
25.     a2.attribute.updateWeights( attribute );
26.     return(``true if points match according to p_match(attribute.distanceTo(a2.attribute))'');
27.   }
28. }
```

Figure 7.1: Text agent implementation.

chapters. This observation results in highly effective classifications, considering its simplicity. It also, unfortunately, introduces a global comparison function that cannot easily be used in our decentralized setting. The notion that relative word frequencies distinguish content bearing from unimportant terms is a general principle underlying text clustering methodologies. Thus it turns out that decentralizing text clustering requires not only that we decentralize the methods of determining a classification and placing data items correctly within it, but that we also decentralize the way in which comparison measures between data items are determined.

7.2 Model

In this chapter we do not change the overall behavior of the agents from Chapter 6, only the data type of agent attributes and the manner in which matches are determined. Figure 7.1 thus gives new pseudo-code for only the Agent and Point objects. Since we study a possible application for clustering agents, we also switch to using the weak synchronization version of timing discussed in Section 2.5.1. We, however, leave the description of how this timing model changes the Clique object until Chapter 8. The full pseudo-code for the Cliques used in these experiments can be found in Figure 8.1.

In order to test the ability of the agents to cluster text, we consider a 100-point data set containing the first ten chapters of each of ten books, listed in Figure 7.2, chosen to have distinct subjects. These chapters are between 343 and 15733 words long. Agents are each

weigting function	similarity coeffiecint	number of clusters			points out of place		
		min	avg	max	min	avg	max
none	Dice	11	15.46	25	4	10.76	27
≈IDF	$\sum_{k=1}^{m} \min(t_{ik}, t_{jk})$	10	13.5	19	5	11.1	19
≈IDF	$\sum_{k=1}^{m} (t_{ik} + t_{jk})$ when $t_{ik} \neq 0, t_{jk} \neq 0$	10	13.5	19	0	6.58	17

Table 7.1: Results of text clustering agents, 50 trials, each run for 500 turns, per experiment.

given the entire text of a chapter as their data point. Following the word vector model agents process these texts to create a vector of all unique words (without stopping or stemming). We represent this word vector as $T = \{t_1, \ldots, t_m\}$, where there are m unique terms in the entire document set, though of course agents only know of the terms their text contains and thus have a 0 weight for all other terms.

There are a number of standard methods of assigning term weights and calculating word vector similarity. Table 7.1 shows data for clusters found using three different combinations of weighting schemes and distance functions. In the first of these all words in the text are given a weight of 1 and we use the Dice coefficient:

$$D(T_i, T_j) = \frac{2(\sum_{k=1}^{m} t_{ik} \cdot t_{jk})}{\sum_{k=1}^{m} t_{ik} + \sum_{k=1}^{m} t_{jk}}$$

This gives a measure of the number of words two texts have in common, normalized by the lengths of the texts. We see in Table 7.1 that even with this simple method agents are already able to find passable clusterings.

A more advanced method of weighting terms is to use each word's inverse document frequency:

$$IDF(t_k) = \log\left(\frac{N}{n_k}\right) + 1$$

where N is the total number of documents, and n_k is the number of documents in which term t_k appears. This weighting method is based on the assumption that words that occur very frequently, or almost never, among the documents are likely to be functional words, while those with a middling frequency are likely to be content bearing. We see here a problem, however, in that in order to calculate the inverse document frequency of a word an agent must examine the entire document set. Since this is not possible in a decentralized setting we instead have agents attempt to estimate inverse document frequencies by observing the frequency of words in the neighbors with which they attempt to make matches:

$$\approx IDF(t_k) = \log\left(\frac{\text{total matches attempted}}{\text{potential neighbors that contained term } k}\right) + 1.$$

To account for differences in document lengths agents normalize their word weights so that their word vector sums to 1. Agents then use one of the simpler methods of determining similarity, namely the sum of the minimal weight of each term:

$$\sum_{k=1}^{m} \min(t_{ik}, t_{jk}).$$

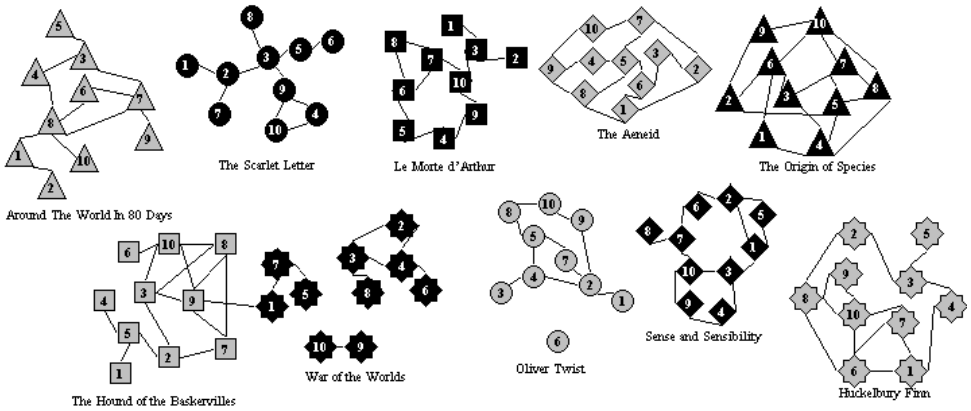


Figure 7.2: Example clustering of text data set.

Alternatively, we find in row three of Table 7.1 that using the sum of all weights for each common term works slightly better:

$$\sum_{k=1}^{m}(t_{ik}+t_{jk}) \text{ when } t_{ik}\neq 0, t_{jk}\neq 0.$$

Since changing the weights of words changes link strengths, and thus requires updating data held by the clique, we have agents reevaluate weights only after each 1000 attempts to make a match.

7.3 Clusters Found

Figure 7.2 shows an example of an average clustering found using the last of the three methods in Table 7.1. The clustering divides the data set into 12 clusters and places 6 points incorrectly. From the diagram we can see that this is actually quite a reasonable clustering of the data set. The mistakes that are made are similar to those seen for spatial data, the "War of the Worlds" cluster has been broken up, one of its components has joined "The Hound of the Baskervilles" and one of the "Oliver Twist" points has gotten lost. Considering that agent clusterings are dynamic these mistakes are most likely to be temporary, so we expect that over time the average clustering will be the correct one.

Table 7.2 shows for three of the clusters from Figure 7.2 the top 25 words, scored by adding the word weights of all agents in the cluster that contain that word. In addition, the table shows the scores and ranks these words would have if all agents used the actual inverse document frequency of the words for this data set as their weights. We see from this table that the agents were able to make reasonably good estimates of the central inverse document frequency values.

	Around The World In 80 Days (2831 words total)				La Mort d'Arthur (979 words total)				Sense and Sensibility (2506 words total)			
agent rank	word	agent score	actual rank	actual score	word	agent score	actual rank	actual score	word	agent score	actual rank	actual score
1	fogg	0.034	1	33.03	unto	0.034	1	30.79	dashwood	0.037	1	33.03
2	phileas	0.034	2	33.03	knights	0.033	5	25.47	norland	0.032	3	27.88
3	bombay	0.028	3	25.47	uther	0.032	6	25.47	elinor	0.031	2	27.88
4	suez	0.028	5	25.47	merlin	0.032	3	27.88	marianne	0.028	7	25.47
5	passport	0.027	8	22.64	king	0.030	8	23.86	park	0.028	4	27.18
6	passepartout	0.026	4	25.47	arthur	0.030	2	28.78	handsome	0.028	5	26.51
7	p.m	0.025	7	22.64	brastias	0.030	4	25.47	marianne's	0.027	8	25.47
8	mongolia	0.022	17	19.98	ulfius	0.027	10	22.64	barton	0.026	11	22.64
9	club	0.022	6	23.12	wherefore	0.026	7	24.64	john	0.025	6	26.09
10	steamers	0.022	18	19.98	barons	0.025	9	22.64	mrs	0.024	9	23.86
11	steamer	0.022	9	22.64	lords	0.023	16	17.42	middleton	0.024	12	22.64
12	october	0.022	12	20.91	archbishop	0.023	12	19.98	income	0.024	20	19.98
13	repaired	0.021	20	18.86	kay	0.023	14	19.98	wishes	0.023	14	22.33
14	visaed	0.021	31	16.88	ye	0.022	11	21.48	daughters	0.022	10	23.57
15	departure	0.020	23	18.20	tintagil	0.022	24	16.88	dashwood's	0.022	19	19.98
16	eighty	0.020	16	19.98	igraine	0.022	13	19.98	propriety	0.021	21	19.98
17	robber	0.020	15	20.39	wales	0.020	48	13.49	situation	0.021	13	22.33
18	reform	0.020	10	21.83	pentecost	0.020	44	13.49	conversation	0.020	23	19.54
19	detectives	0.020	38	15.98	ector	0.019	21	16.88	edward	0.020	26	18.86
20	brindisi	0.020	28	16.88	purvey	0.019	23	16.88	cottage	0.019	15	21.56
21	paris	0.019	21	18.48	merlin's	0.019	41	13.49	affection	0.019	17	20.62
22	consulate	0.019	72	13.49	canterbury	0.018	18	16.88	herself	0.019	16	20.99
23	quay	0.019	80	13.49	pendragon	0.018	22	16.88	dashwoods	0.018	32	16.88
24	detective	0.019	24	17.42	nourish	0.018	43	13.49	elinor's	0.018	33	16.88
25	doesn't	0.019	39	15.98	assay	0.018	36	13.49	required	0.018	27	18.27

Table 7.2: Top 25 words from three example clusters.

7.4 Conclusions

In this chapter we showed how agents could cluster unstructured text data, indicating that our agent clustering procedure can handle fairly complex data types. These experiments also demonstrate how the abstract agents can be modified fairly easily to work with new forms of data. Moreover, we showed how an essentially global distance measurement, a word's inverse document frequency, could be adapted for use in a decentralized setting.

Since agents each separately estimated word document frequency they no longer had a single common metric to measure the distance between points. Interestingly, in spite of this imprecise estimation they were still able to correctly cluster points. This gives an indication that agents can, as we had hoped, have individual opinions about the similarity of points, yet still be able to agree on clusters.

The experiments in this chapter were on a fairly small data set. For a much larger data set the speed with which words' inverse document frequencies are learned, and thus the speed at which clusters form, could be affected. It may be the case, however, that for larger data sets the learned estimations of distance will simply be less precise. If agents are able to deal with even less agreement between individuals' perception of distance, they should be able to find clusters in spite of this. Nonetheless, in the future, we need to extend these experiments to larger data sets to confirm these conclusions.

Chapter 8

Adaptive Clusters

In Chapter 6 we chose to to fix the maximum size to which cliques could grow. Though we found that cliques can adjust their size to nearby actual data cluster sizes, overall this limit restricts the range of data clusters that can be identified by the agents. In this chapter we consider a manner in which agents can learn cluster area and density, thus allowing them to perform on a much broader range of data sets without having to adjust parameters. The work in this chapter was originally presented in [27] and [29].

8.1 Introduction and Related Work

Clustering in general is a difficult problem because clusters can have many unknown characteristics, such as size and shape, which make them difficult to delineate. Since the definition of optimal clusters is imprecise, clustering algorithms need to include some specification of how cluster boundaries should be determined. For decentralized clustering this is even more problematic than for centralized algorithms because global information such as the size or range of a data set is unavailable. Moreover, the notion of autonomy can exclude the use of a single agreed comparison function. By decentralizing clustering we thus exacerbate the problem of how to tell a computer the characteristics of the clusters that we wish to find.

In general, one of the great difficulties encountered when designing clustering algorithms is defining the term "cluster" in a way that a computer can interpret. Intuitively a cluster can be seen as a connected dense region of points, surrounded by a less dense region. Finding clusters is thus finding boundaries between density regions. Because these boundaries are generally fuzzy, computer algorithms need a more concrete definition. To achieve this, additional restrictions are used to make clusters calculable. The k-means algorithm [20], for instance, fixes the number of clusters in the data set and, hidden in its squared error measure, makes the assumption that clusters are roughly spherical. Given this information, clusters will extend to their natural boundaries, provided that centoids are chosen correctly. The k-means passes and various starting heuristics are methods of estimating the "correct" centroids.

Hierarchical algorithms [15] (we will use the top-down minimal spanning tree algorithm as an example) split clusters along the largest existing edge between points that are currently in the same cluster to obtain each level. By doing this they in essence say "if there is a density boundary, this edge must cross it." This method identifies boundaries nicely, but leaves the problem of choosing which level of the tree contains the correct clustering. By

basing this on some statistic, like the total squared error or the number of clusters, additional assumptions are introduced that are, again, dependant on the data set.

Density-based clustering specifically looks for density boundaries by walking through the data set from point to nearest point. Here it is clear to see that a definition of a boundary is required. DBSCAN [8] specifically says that clusters are areas with a given minimal density. DBSCLADS [52] makes the assumption that clusters are of roughly uniform density, with a parameter to define "roughly."

Overall, the standard data set dependent "clues" used by clustering algorithms include:

- the number of clusters, or analogously, the expected cluster size,
- the minimal gap between clusters,
- the minimal density of clusters.

Each of these gives a global standard for the data set, but can be used in decision functions locally. This allows centralized clustering algorithms to be parallelized, provided that global information about the data set can be shared between processing units [34].

In the previous chapters we showed how clustering could be performed by a decentralized multi-agent system, given that we could set beforehand the maximum size of the clusters we wished to obtain. In fact, knowing any of the above clues correctly makes the form of decentralized clustering that we studied fairly easy, as it gives clusters an indication of when they have reached their boundaries. The real challenge occurs when these clues are absent, or do not hold for all clusters in the data set.

In this chapter we show how cliques that lack any central coordination between them can learn cluster parameters by watching their internal composition over time. This allows them to find clusters within data sets with varying cluster sizes and densities, and even to identify clusters with widely differing characteristics within the same data set. Thus, whereas in Chapter 6 we study the problem of clustering in a fully decentralized way across possibly large networks, in this chapter we study the problem of dynamically determining when to continue or to stop clustering.

8.2 Model

In this chapter we modify the timing of the agent simulation from the central clock model to a more realistic weak synchronization model based on agent actions (as described in Section 2.5.1). Figure 8.1 gives pseudo-code. The behavior of these agents is basically the same as those in Chapter 6. The only difference is that Agent and Clique procedures are now triggered by calls from other agents or internal conditions, in place of clock signals. As before link swapping is achieved by the *serachForMatches* process in each agent trading in links via the Clique *tradeIn* procedure. Now, however, the **wait until** clause in *tradeIn* (line 26) does not wait for a clock phase, but instead simply counts the number of links that have been traded in. Since the clique map includes the status of clique links, a clique will know when all of its currently nonmatching links have been received, and can return new neighbors at this point. In practice we relax this slightly so that *tradeIn* returns neighbors after 40% of nonmatching links have been traded in. This is because agents are no longer

synchronized with each other, so that the communication necessarily to check if two agents match, in line 18, could take some unknown amount of time. Thus it is best that a clique does not move at the rate of its slowest agent or neighbor, but rather waits only until it has enough links that new neighbors are likely to be different than previous ones.

Forming and downgrading connections within a clique is still achieved through the process *serachForConnections* (lines 29-41) which creates new connections, and the procedure *reconsiderComposition* (lines 47-60) which picks unprofitable matches and connections to downgrade. These procedures act on the principle that all connections that can be made should be made, until the clique's limits have been reached, at which point steps should be taken to ensure that clique assets are used optimally. Thus when better matches are found that cannot be made, bad connections should be broken to make room in the clique while all other matches are unneeded and should be discarded to free links to continue searching for better ones.

The *searchForConnections* process is governed by a **whenever** clause in line 30, which pauses operation until a "sufficient" number of matches have been found. The purpose here is to synchronize with the rate at which agents search for matches, approximating the underlying assumption that the matches considered are the best currently available among the clique's neighbors. If new connections are chosen too quickly they will only have to be broken again, which is a fairly costly process since it involves many changes to the clique composition. In practice we implement "sufficient" as "each time the trade in process returns new acquaintances." The additional "no composition changes pending" condition is used to ensure that a clique can be involved in negotiating only one merge or split operation at a time. This simplifies the clause in *requestConnection* which determines whether a connection request should be accepted (line 43). This clause now becomes a straightforward check of the resulting size of the clique, should the connection be formed.

When a connection request fails both clusters involved call *reconsiderComposition* to check whether they in fact make the best use of available links. In principle, the worst link will be downgraded. A deterministic choice can however result in a race condition where a match is repeatedly upgraded and downgraded, protecting a slightly better link that could be broken to more profit. Thus the choice clauses in lines 52 and 57 are used to introduce some degree of randomness in choosing not the worst link but among all of the not-so-good links. For this weighted random choice we employ the heuristic, used in previous versions, described in Section 5.4.

8.3 Effects of Learning The Matching Range

In Section 5.4, we have already studied one form of cluster adaptation, learning the expected distance to nearby points on a line. For clustering 2-dimensional points in Chapter 6 we simply carried over the same learning behavior. For completeness, in this section we briefly review this behavior to ensure that it also works well for 2-dimensional points.

Figure 8.2 presents E^2 over time for agents with a fixed matching range (t) of 5, compared with agents which learn t. The non-learning agents use the following function to decide matches:

```
1.  object Port{
2.    neighbor:Port; matching:Boolean; connecting:Boolean; myAgent:Agent;
3.    internal procedure reset(){
4.      connecting := FALSE; matching := FALSE;
5.      neighbor.reset();
6.    }
7.    internal procedure linkDistance(){
8.      ``Euclidean distance between myAgent.point and neighbor.myAgent.point''
9.    }
10. }
11. object Agent{
12.   acquaintances: internal set of Port; //Agent's view of its links
13.   c: Clique; //The clique of which this agent is a member
14.   point: [Real, Real]; //This agent's attribute
15.   process searchForMatches [for each aq in acquaintances]{ //One process for each acquaintance
16.     whenever ( aq.matching = FALSE ){
17.       aq.neighbor := c.tradeIn( aq.neighbor );
18.       ``negotiate match with new neighbor'';
19.     }
20.   }
21. virtual object Clique {
22.   members: set of Agent; //All agents that are members of this clique
23.   unwantedAcquaintances: set of Port;
24.   external procedure tradeIn( p:Port ) returns ( p':Port ){
25.     unwantedAcquaintances.add( p );
26.     wait until( ``sufficient unwanted acquaintances have been received'' );
27.     return unwantedAcquaintances.remove( ``choose an unwanted acquaintance to return'' );
28.   }
29.   process searchForConnections{
30.     whenever (``sufficient matches have been found'' & ``no composition changes pending''){
31.       worstConnection:Port := members.all.acquaintances.max( linkDistance(), ( connection = TRUE ) );
32.       bestMatch:Port := members.all.acquaintances.min( linkDistance(), ( matching = TRUE & connecting = FALSE ) );
33.       if ( members.contains( bestMatch.neighbor ) ){
34.         if ( bestMatch.linkDistance() < worstConnection.linkDistance() ){
35.           ``upgrade bestMatch into a connection'';
36.         }else ``downgrade bestMatch to a nonmatching link'';
37.       }else if ( bestMatch.neighbor.c.requestConnection( self ) = ACCEPT ){
38.         ``upgrade bestMatch into a connection'';
39.       }else reconsiderComposition();
40.     }
41.   }
42.   external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE)
      when (``no composition changes pending''){
43.     if (``proposed clique does not exceed current clique limitations''){
44.       return ACCEPT;
45.     }else { reconsiderComposition(); return REFUSE;}
46.   }
47.   internal procedure reconsiderCompositon(){
48.     worstConnection:Port := members.all.acquaintances.max( linkDistance(), ( connection = TRUE ) );
49.     bestMatch:Port := members.all.acquaintances.min( linkDistance(), ( matching = TRUE & connecting = FALSE ) );
50.     if ( bestMatch.linkDistance() < worstConnection.linkDistance() ){
51.       candidates: set of Port := ``all clique Ports for which matching = TRUE '' ;
52.       p:Port := ``choose a member of candidates, weighting by link distances'' ;
53.       p.reset();
54.       if (``clique is no longer connected''){``split off a new clique'';}
55.     }else {
56.       candidates: set of Port := ``all clique Ports for which matching = TRUE & connecting = FALSE '' ;
57.       p:Port := ``choose a member of candidates, weighting by link distances'' ;
58.       p.reset();
59.     }
60.   }
61. }
```

Figure 8.1: High level agent and clique implementation.

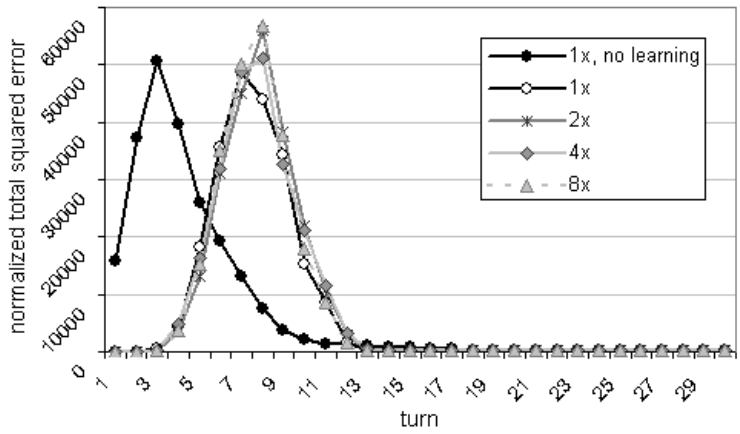


Figure 8.2: Normalized total squared error over time for learning agents and stretched data sets.

```
[Agent] external procedure negotiateMatch( other: Agent ) returns ( ACCEPT, REFUSE ){
  distance: Real := ``Euclidean distance between this agent's and other's data points'';
  if ( ``true with probability 1 − distance/t '' & other != self ){
    return ACCEPT;
  }else return REFUSE;
}
```
(8.1)

while the learning agents use the learning rule from Section 5.4:

```
[Agent] t: Real := 0;
[Agent] external procedure negotiateMatch( other: Agent ) returns ( ACCEPT, REFUSE ){
  distance: Real := ``Euclidean distance between this agent's and other's data points'';
  updateMatchingRange( distance ); //Possibly increase t
  if ( ``true with probability 1 − distance/t '' & other != self ){
    t := distance; //Decrease t so that next match found will be an improvement
    return ACCEPT;
  }else return REFUSE;
}
```
(8.2)

```
[Agent] target: Real; step: Real :=0; count: Integer := 0;
[Agent] internal procedure updateMatchingRange( distance: Real ){
  if ( distance < t ){ count := 0; step := 0 } //t is large enough
  if ( count++ < 50 & step = 0 ){ target := minimum( target, distance ); } //Watch a series of distances
  else if ( step = 0 ){ step = (target − t)/100; } //Slowly increase t to the smallest distances seen
  else if( t < step ){ t = t + step; }
  else { step := 0; count := 0; } //Restart loop
}
```

The data sets "1x, no learning" and "1x" compare these two agent versions clustering the data set shown in Figure 8.3. Since t essentially defines the expected density of clusters, Figure 8.2 also shows curves for the same data set stretched in both the x and y directions by factors of 2, 4 and 8. To account for the difference in distance between agents in these clusters, these curves were normalized by dividing E^2 by 2^2, 4^2, and 8^2 respectively. As can be expected, the learning agents take longer to converge to the correct clustering since they require some time to adjust t. On the other hand, the fact that all the curves converge to the same value shows that in all cases the correct clustering was found. This implies that the agents discover t correctly.

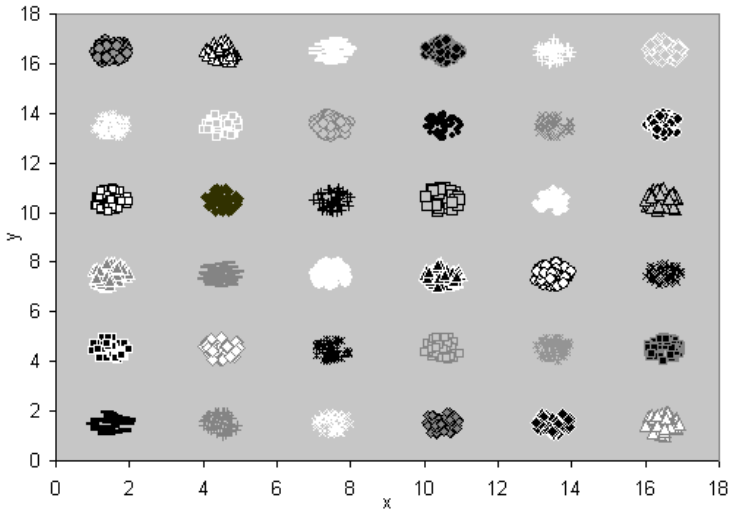


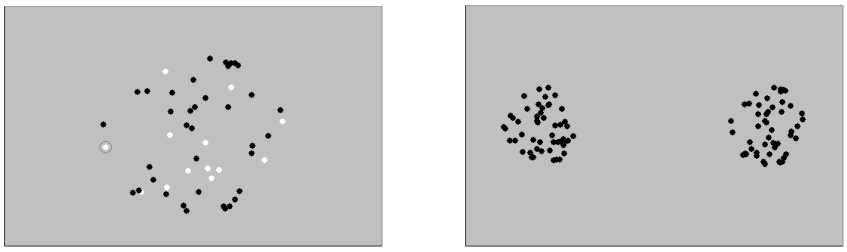
Figure 8.3: A simple data set containing 36, 50-point clusters (point location) and the agent clusters found (point coloring) using non-learning agents, $t = 5$.

8.4 Learning the Maximum Cluster Size

While agents can relatively easily learn a suitable matching range, the correct clique size is more difficult to determine by only observing the current surroundings. Where one cluster begins and another cluster ends depends on all of the points in the boundary area, and is often a matter of opinion. Learning S, the maximum size to which cliques can grow therefore requires a more complex heuristic.

Let us hypothesize that cliques have a function, *shouldSplit*, that returns true if they are made up of more than one data cluster, and false otherwise. Adjusting S then becomes straightforward. A clique first grows to some initial S. It then waits until its membership becomes stable. The black points in Figure 8.4 shows us two examples of possible configurations of the clique at this point. In 8.4(a) the clique contains points from a single data cluster and *shouldSplit* will return false. In 8.4(b) the clique covers points in two separate data clusters and *shouldSplit* will return true.

Thus once a clique's membership is stable, if *shouldSplit* is true the clique knows that it has grown too large and should break into two, resetting S for each of the resulting cliques. These cliques can then wait until their composition stabilizes and repeat the process to provide further adjustments. On the other hand, if *shouldSplit* returns false the clique only knows that it is either the correct size or too small, and thus needs to check if it should grow. To do this it can simply run *shouldSplit* again, considering its nearest matching point to also be part of the clique. If *shouldSplit* still returns false the clique knows there is at least one more point that it should include and thus it should increase S. A call to a Clique procedure *reconsiderSize*, which implements this behavior, can be inserted in the *reconsiderComposition* function in Figure 8.1 before line 48.



(a) (b)

Figure 8.4: Example good clique (a) versus a clique that covers two data clusters (b). Black points are clique members, white points are unfound points in the data cluster. The circle in (a) marks the current nearest matching point.

```
[Clique] internal procedure reconsiderSize(){
  if ( ! isStable() ) return; //Wait until the clique composition stabalizes
  unwantedLinks: set of Port := shouldSplit(); //Returns Ports of links that should be broken to correctly split the clique
  if ( unwantedLinks.size() > 0 ){ //The clique is too large
    ''downgrade all unwantedLinks to nonmatching'';
    while''the clique members are not connected''{''split off a new clique''; }
  }else { //Check if next connection to be made would also fit in the clique
    ''temporarily add the agent that the best matching link connects to to the clique'';
    if shouldSplit().size() = 0 { ''increase S''; } //If the resulting clique is still good increase S
    ''remove the temporary addition'' ;
  }
}
```
(8.3)

The growth process achieved through *reconsiderSize* is synchronized by the *isStable* condition. A clique could determine if its composition has stabilized by recording and analyzing its membership over time. We however opt for the simpler method used in other synchronization procedures: just wait for a long time. In this case we implement cliques to call *reconsiderSize* with a probability based on the clique size, $|c|$: $p(|c|) = 5/|c|$. Again, mistakenly growing a clique is not a problem in terms of correctness since the clique will simply be split again at a later time, though it does effect performance since cliques that are allowed to grow too quickly can engulf several data clusters before being correctly split.

Figure 8.5 shows the clusters found by these improved agents for a data set containing clusters of varying density, ranging from 20 to 200 points in size, with S initially set to 10. These agents were given a "perfect" *shouldSplit* function which used the knowledge that the minimum distance between two clusters in this data set is 2.0 units:

```
[Clique] minInterClusterDistance: Real = 2.0;
[Clique] internal procedure shouldSplit() returns set of Port{
  //Returns connections that should be broken to split this clique correctly
  unwantedLinks:set of Port :=
  ''all clique ports for which linkDistance() ≥ minInterClusterDistance'';
  return unwantedLinks;
}
```
(8.4)

Figure 8.5 shows that though they no longer find perfect clusters, these new agents are now able to learn the correct size of the data clusters. There are a couple of typical mistakes. At point A two data clusters have been combined. This has occurred because one data cluster is much larger then the other. When cliques split, the new S values were reset to 1.5 times their resulting size, giving the new cliques some room to grow should

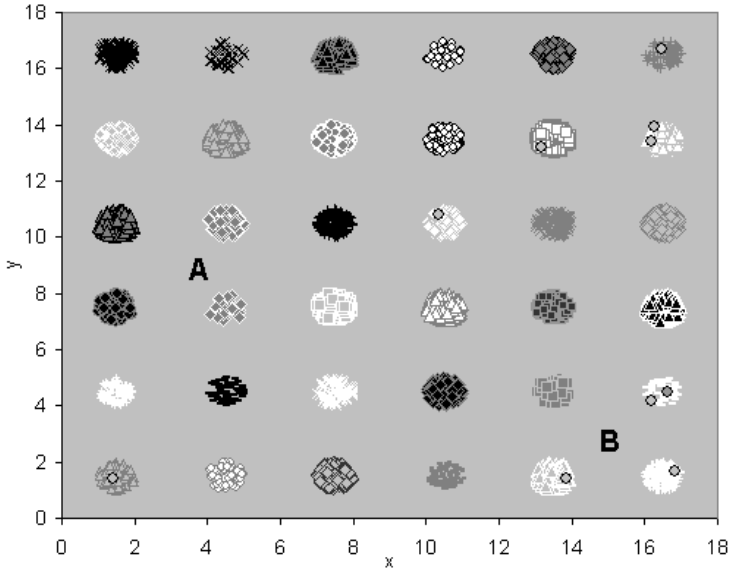


Figure 8.5: Learning agents' clustering of a data set with clusters of varying density. Grey circles are single agents.

they be missing points. This extra room makes a large difference to the speed at which cliques form, but allows very small cliques to be repeatedly added on to then again split off of neighboring large ones. Figure 8.5 also indicates, as at point B, that some single agents tend to become "lost" by the system during the initial clique formation phase. Once good cliques have formed, these lost agents eventually find their correct clique, but do so slowly since they are quickly rejected from any incorrect clique they join. This also occurred with the non-learning agents, though less often.

Unfortunately the "perfect" *shouldSplit* function, (8.4), requires a-priori knowledge of the intra-cluster distance. On the other hand, the *shouldSplit* function essentially calculates an internal clustering of a clique's points. Thus in theory (8.4) could be implemented using any existing clustering algorithm:

```
[Clique] internal procedure shouldSplit() returns set of Port{
  c1:clustering := ''Clustering in which all members are in a single group'';
  c2:clustering := ''All members clustered into two groups''
  if ( evaluate( c2 ) > evaluate( c1 ) ){ //If c2 is better then c1 according to some measure
    return ''all connections that are between the two groups in c2'';
  else return ∅;
}
```
(8.5)

In practice however the inability to accurately compare clusterings of a data set containing different numbers of clusters makes this difficult. For instance, if we were to use the E^2 measure to implement *evaluate*, E^2_{c2} will always be slightly less then E^2_{c1}, due to E^2 preference for many clusters. Thus a parameter, γ, needs to be introduced to govern just how much worse E^2_{c2} must be before a clique is split. There is, however, no value for γ that

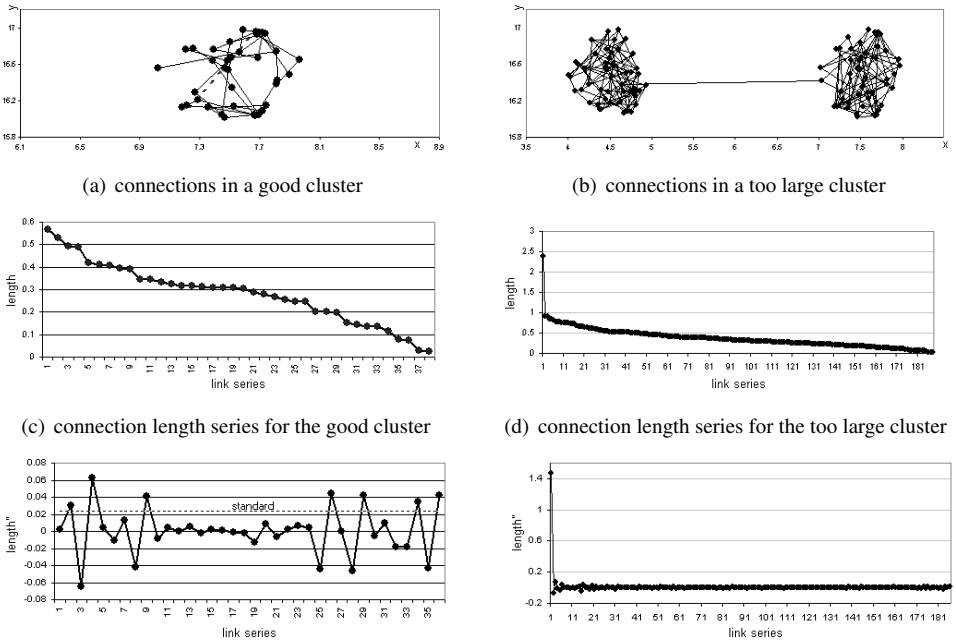
is appropriate in all cases since as cliques become larger and closer together a few incorrect points included from a neighboring data cluster will make much less of a difference to the E^2 of the clique as a whole. In fact, experimentally we were unable to find a value of γ that produced correct clusters even on the straightforward data set in Figure 8.5. A value large enough to allow cliques to grow also allowed very large cliques to form, which eventually grew to engulf the entire data set. A lower value, on the other hand, split cliques too easily. Thus, more advanced implementations of *shouldSplit* must be explored.

Overall, however, we have succeeded in turning a large distributed clustering problem into many easier small central ones. Since there is no perfect measure of a good clustering other versions of *shouldSplit* are likely to also require some sort of γ parameter. Nevertheless, γ is far better then the density and clique size parameters we have used up to now, since it allows agents to cluster a much larger range of data sets without adjustment. In the following section we will examine one possible implementation of the *shouldSplit* function, and investigate the range of data sets it can cover.

8.4.1 Determining Cluster Boundaries

In order to create a suitable implementation of *shouldSplit* we need to reconsider the clustering characteristics stated as a goal in the introduction. Since *shouldSplit* operates on small, known sets of agents there is no need for it to be distributed. However, there were other reasons to avoid centralization which still need to be addressed. These have to do with the fact that a central clusterer may need a large amount of information about the data points and how to compare them, something better left up to the agents. For this reason, a version of *shouldSplit* that does not require knowledge of the actual data points or the ways in which they were compared when choosing links is desirable. Ideally, *shouldSplit* should use only the data that is already collected by the clique to run the other steps in the clustering procedure: the strength of each of the connecting and matching links. The process or matching and breaking in the basic agent procedure should create a network of connections that somewhat approximates the minimal spanning tree for a clique. Such a tree represents how well agents are connected to each other and exposes links that should be broken. This fact, and the fact that connections between two data clusters are in most cases exceptionally long compared to connections that join agents in the same data cluster, can be used to discover inter-data-cluster connections.

In the remaining experiments we use a version of *shouldSplit* that follows this approach. To determine which connections really belong in a clique we examine a series containing the clique's connections, ordered by the distance between the agents each connects. These connection lengths, within a real cluster, will be similar but not identical. Thus we expect this list of lengths to change gradually, while a gap between clusters should be indicated by a sudden change. Such a gap can be detected by estimating the second derivative of the connection length series: $f''(x) \approx (y_2 - 2y_1 + y_0)$, where y_2, y_1 and y_0 are consecutive connection lengths. Sudden changes in connection length will appear as large peaks in this second derivative series. On the other hand, again due to the variation of connection lengths in real clusters, even correct clusters will exhibit peaks. To determine what "large" is we calculate the standard deviation of the second derivative series, leaving out the highest and lowest points (since a long link between two data clusters can out shadow all others



(a) connections in a good cluster

(b) connections in a too large cluster

(c) connection length series for the good cluster

(d) connection length series for the too large cluster

(e) second derivative of 8.6(c), good cluster

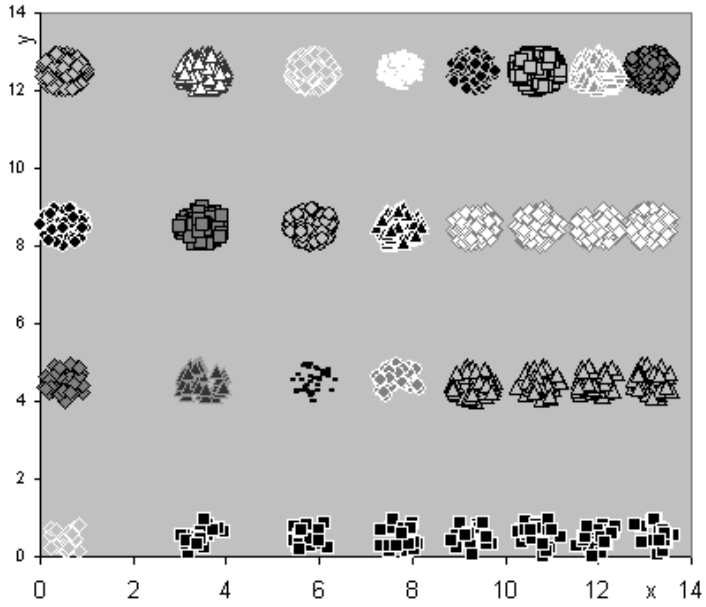
(f) second derivative of 8.6(d), too large cluster

Figure 8.6: Example *shouldSplit* calculation for a good and a bad clique

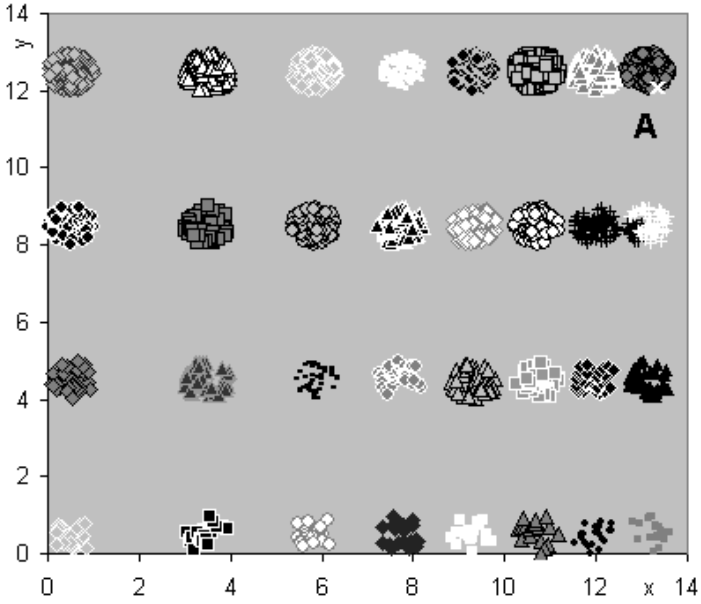
when clusters are small, decreasing the accuracy): $S_{N-1} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$. The error parameter γ is then used to define how many times larger than the standard deviation a peak must be before we choose it as a splitting point for the clique. When we find a peak we break the corresponding connection. The resulting calculation, given below, is illustrated in Figure 8.6 for a correctly sized and a too large clique.

```
[Clique] internal procedure shouldSplit() returns set of Port{
  lengthSeries: set of distance := ''linkDistance() for each connection'';
  lengthSeries.sortDecreasing(); //Fig 9(c) & 9(d)
  lengthSeries'': set of Real:= estimateSecondDerivate( lengthSeries );//Fig9(e) & 9(f)
  stdDev: Real:= standardDeviation( lengthSeries'' ); //Fig 9(e) dotted line
  //find the unwanted links, actually only returns first one encountered
  breakPosition: Real := ''first member, m, in lengthSeries" for which m≥ stdDev×γ '';
   if breakPosition = null return ∅
  breakLength:Real := ''member of lengthSeries corresponding to breakPosition in lengthSeries"'';
  return ''any one connection with a linkDistance() of breakLength'' ;
}
```
(8.6)

Figure 8.7(a) shows the resulting clusters using $\gamma = 5$ for a data set containing rows of clusters placed increasingly close together, and with increasingly more points per cluster in each row. There are several factors that influence how easily *shouldSplit* distinguishes clusters. The most important of these is the ratio of the distance separating points within a cluster to the distance between clusters. The bottom row in figure 8.7(a) contains clusters with 20 points each. With such sparse clusters small gaps between clusters easily look like part of the cluster. Thus only the most separated cluster, on the far left, is distinguishable



(a) All connections kept



(b) Only minimal spanning tree of connections kept

Figure 8.7: Test of agent's ability to distinguish clusters for varying density/gap-between-clusters ratios.

to the agents. The next rows up contain clusters of the same area with 40, 80 and 160 points each. As the cluster density is increased small gaps between clusters become more distinct and it can be seen that the agents become better able to separate the clusters from one another.

The *shouldSplit* function described in (8.6) makes the assumption that the matching and breaking process will keep to a minimum the number of connections between data clusters. For this reason, only the single connection before a peak in the second derivative series is broken when an overgrown clique is found.[1] This leads to some inaccuracy in separating clusters since often two or three connections can be made between data clusters within a single clique. One possible fix for this is to remove all cycles in the connections graph within each clique. This limits a clique to only containing the minimum number of connections to keep it intact. As a result breaking any connection will break up the clique. To implement this behavior we add functionality that, whenever an internal connection is created, finds the resulting cycle and breaks its longest connection: [2]

```
[Clique] process searchForConnections{
  whenever (``sufficient matches have been found'' & ``no composition changes pending''){
    worstConnection:Port := members.all.acquaintances.max( linkDistance(), ( connection = TRUE ) );
    bestMatch:Port := members.all.acquaintances.min( linkDistance(), ( matching = TRUE & connecting = FALSE ) );
    if ( members.contains( bestMatch.neighbor.myAgent ) ){
      if ( bestMatch.linkDistance() < worstConnection.linkDistance() ){
        ``upgrade bestMatch into a connection'';
        "find the worst connection in the resulting cycle and downgrade it to unmatching";
      }else{
        ``downgrade bestMatch to a nonmatching link'';
        "repeat this whenever clause";
    }else if ( bestMatch.neighbor.c.requestConnection( self ) = ACCEPT ){
      ``upgrade bestMatch into a connection'';
    }else reconsiderComposition();
} }
```
(8.7)

Figure 8.7(b) shows this new version of the agents, using $\gamma = 7$, run on the data set from figure 8.7(a). It can be seen that the agents are now able to distinguish even very small closely packed clusters. Now, however, because of the lack of cycles, it is more likely that small parts of the clusters get broken off (and eventually reattached) as the set of found connections improves. This behavior occurs in the upper right hand corner at the points represented by the white x's near label A.

8.4.2 Choosing γ

Ideally, the most suitable value of γ would be independent of the number of points in a cluster. Unfortunately, when points are placed randomly within clusters, the second derivative series will always show some amount of random variation. This variation can be large, and the longer the connection series is, the more likely we are to see a random fluctuation that results in an incorrect decision to split a clique. This means that the more agents a clique contains, the more likely a small γ is to break it incorrectly. Thus γ must be set low to accurately distinguish low density clusters that are very close together, but at the same time needs to be kept high enough to avoid accidentally splitting larger clusters. Because of this dichotomy the value of γ is not wholly independent of cluster size. However, because clique

[1] If instead all longer connections were broken agents would have trouble with sparse clusters located next to dense clusters

[2] Since this results in cliques containing many more internal matches we also need to modify the connection creation behavior to repeatedly make the best matches into connections until it gets to an external connection, rather then just upgrading the single best match.

sizes are already limited by coordination costs, the value of γ used only needs to cover a sufficiently large range of sizes. A range between 10 and 500 points could be reasonable for many applications. For instance, this is a good range for the number of results that should be returned by a search query, if we were to use cliques to define replies.

In order to obtain a more accurate picture of the effect of γ, we examine a series of experiments using a data sets containing two data clusters with varying internal densities, and a varying separation between them. Depending on the value of γ we observed one of the following behaviors occurring in each trial:

γ is set too low: the clusters are broken into many smaller pieces.

γ is set correctly: the agents easily distinguish the two clusters, though intermittently a few single agents can split off from and later reattach to the cluster edges.

γ is set too high: the clusters are joined into a single large clique.

γ is slightly lower than the correct value: the clusters are repeatedly found correctly, broken into pieces, then found again.

γ is slightly higher than the correct value: a single clique is formed, broken up, (often correctly, or but also incorrectly) and formed again.

Table 8.1 summarizes this experiment, run using the original version of *searchForConnections* (Figure 8.1). Trials are divided into three categories based on the number and size of cliques found after the trial had been run for a fixed amount of time. The categories are: "correct", when at least 93.75% of agents were placed in their correct clique, "joined", when at least 93.75% of agents were placed in a single clique, and "crumbled" when several smaller cliques were formed. From this single time point measurement of the clique sizes it was not possible to distinguish the case where γ was too small from the cases where it was slightly too small or slightly too high. It is interesting to note, however, that if agents could distinguish these cases by observing joining and splitting behavior over time they would have a basis from which to learn a value of γ that fit well with their data.

Table 8.1 shows the percentage of times a trial resulted in each of the above categories, for a series of 100 trials in each category. We used round clusters with a fixed radius, containing random points generated from an even distribution. For each trial we generated a new data set. There are three variables considered, the value of γ which ranged from 3 to 12, the size (density) of each of the two clusters, which ranged from 8 to 512 agents, and the separation between the two clusters. The ability to distinguish clusters depends on the ratio of the expected distance between nearest neighbors within the cluster and the distance between the two clusters. Thus we consider three values of this ratio, 1:1, 1:4 and 1:12. We expect that for the 1:1 ratio the clusters are so close together that they are unlikely to be distinguishable. The 1:4 ratio produces clusters that are fairly close together and thus relatively difficult to distinguish. The 1:12 ratio produces cluster that should be separable. We also expect a lower γ to more accurately distinguish small clusters while splitting up large clusters, while a higher γ will be less accurate for smaller clusters, but handle large clusters correctly.

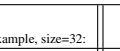
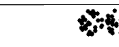
example, size=32:									
separation:	1:1, no gap			1:4, small gap			1:12, larger gap		
	% correct	% crumbled	% joined	% correct	% crumbled	% joined	% correct	% crumbled	% joined
CLUSTER SIZE = 8									
$\gamma = 3$	40	3	57	93	4	3	94	6	0
$\gamma = 6$	13	2	85	78	8	14	89	11	0
$\gamma = 9$	5	2	93	71	1	28	91	9	0
$\gamma = 12$	4	6	90	54	1	45	91	9	0
CLUSTER SIZE = 32									
$\gamma = 3$	19	57	24	56	41	3	63	33	4
$\gamma = 6$	4	0	96	72	4	24	93	2	5
$\gamma = 9$	3	0	97	55	1	44	100	0	0
$\gamma = 12$	1	0	99	38	0	62	100	0	0
CLUSTER SIZE = 128									
$\gamma = 3$	0	99	1	0	100	0	0	100	0
$\gamma = 6$	4	2	94	80	0	20	82	2	16
$\gamma = 9$	0	0	100	42	0	58	92	0	8
$\gamma = 12$	0	0	100	19	0	81	89	0	11
CLUSTER SIZE = 512									
$\gamma = 3$	0	100	0	0	100	0	0	100	0
$\gamma = 6$	3	63	44	29	39	32	55	30	15
$\gamma = 9$	0	0	100	27	0	73	90	0	10
$\gamma = 12$	0	0	100	5	0	95	77	0	23

Table 8.1: Experiments with increasing large pairs of clusters with decreasing gaps between them.

The data in table 8.1 confirms these expectations. In the 1:1 ratio data sets the clusters were fairly indistinguishable. In the 1:4 ratio data sets the number of joined clusters rises as γ is increased, while the number of correctly found clusters grows then falls. Furthermore, the correct clusters are found less often as the cluster size is increased and the best value of γ increases with cluster size. The 1:12 ratio data sets show that for larger gaps clusters of all sizes are usually correctly separated and that a γ of 9 covers our desired size range.

Table 8.2 gives the same experimental data for the version of *searchForConnections* that keeps only a minimal spanning tree of connections (8.7). Here the trends are not as clearly visible. It can be seen, however, that clusters are more easily distinguished for lower denisity/gap ratios, at a cost of a higher tendency to crumble clusters.

Yet another factor that will change the effectiveness of the *shouldSplit* function is the internal distribution of points within a cluster. The clusters examined so far are generated with an even random distribution. The *shouldSplit* function does not dictate that all clique connections be approximately the same length. Instead it assumes that connection length changes gradually. Thus it should also be able to handle other distributions. To confirm this, Table 8.2 also shows data for clusters containing 128 points generated at random with a Gaussian distribution from the center. To compare with the even distribution clusters we set the radius for these clusters at twice the standard deviation of the point distribution function. We find that this uneven distribution does not cause a problem for the *shouldSplit* function, and that the sparseness of points at the edges of the clusters actually makes them easier to distinguish when the separation is small.

8.5 Scalability

There are a fairly large number of factors that affect the speed at which the cliques found by the agents converge to likely data clusters. In particular, we often achieve coordination by simply waiting until it is likely that a particular state has been reached. The exact speed of clustering is also highly dependant on the characteristics of the network the agents

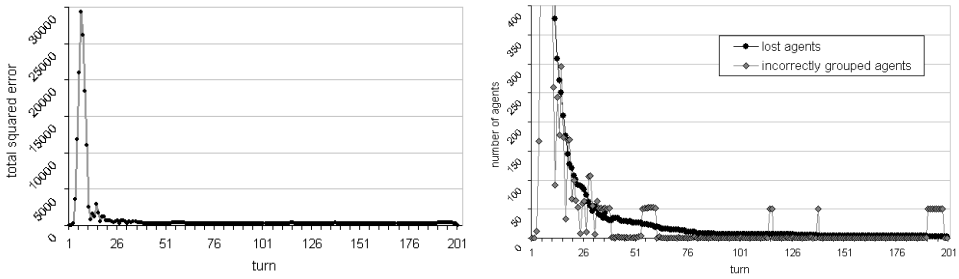
separation:	1:1, no gap			1:4, small gap			1:12, larger gap		
	% correct	% crumbled	% joined	% correct	% crumbled	% joined	% correct	% crumbled	% joined
CLUSTER SIZE = 8									
$\gamma = 3$	63	36	1	78	22	0	78	22	0
$\gamma = 6$	50	43	7	80	20	0	67	33	0
$\gamma = 9$	55	29	16	65	34	1	74	26	0
$\gamma = 12$	55	28	17	65	33	2	80	20	0
CLUSTER SIZE = 32									
$\gamma = 3$	39	60	1	66	34	0	71	29	0
$\gamma = 6$	68	19	13	89	9	2	83	17	0
$\gamma = 9$	56	13	31	91	9	0	95	5	0
$\gamma = 12$	61	16	23	93	5	2	97	3	0
CLUSTER SIZE = 128									
$\gamma = 3$	11	88	1	12	88	0	11	89	0
$\gamma = 6$	48	16	36	100	0	0	100	0	0
$\gamma = 9$	37	10	53	100	0	0	100	0	0
$\gamma = 12$	39	8	53	99	0	1	100	0	0
CLUSTER SIZE = 512									
$\gamma = 3$	0	98	2	2	98	0	0	100	0
$\gamma = 6$	36	45	19	70	30	0	72	28	0
$\gamma = 9$	16	5	79	88	1	11	98	1	1
$\gamma = 12$	5	2	93	69	0	31	89	1	10
GAUSSIAN CLUSTERS, CLUSTER SIZE = 128									
$\gamma = 3$	22	76	2	22	78	0	14	86	0
$\gamma = 6$	96	1	3	97	3	0	100	0	0
$\gamma = 9$	83	6	11	99	0	1	100	0	0
$\gamma = 12$	80	1	19	98	0	2	100	0	0

Table 8.2: MST Experiments with increasing large pairs of clusters with decreasing gaps between them.

communicate over. More important, however, is how the cost of clustering changes as we increase the number of clusters. This section presents, for a fixed implementation, a number of experiments to determine how well the algorithm scales with the size of the data set.

In order to measure how the time to find a clustering changes as the size of the system increases we consider a series of data sets containing an increasingly large grid of clusters. The size, density and spacing of the clusters is kept constant, thus the experiments measure the change in convergence time as the number of clusters in the data set increases. It should be noted that increasing the number of clusters is only one way of increasing the number of agents. If instead the size of clusters is increased different scalability properties will be seen as the internal coordination within cliques outweighs the intra-clique coordination costs. Again, we preclude this issue by limiting this study to small clusters.

Figure 8.8 illustrates how the agent clusterings converge to the underlying data clusters over time. Figure 8.8(a) shows the total squared error over time for the smallest data set in the series. This data set is the basic data set from Figure 8.3. Comparing the E^2 series in Figure 8.8(a) to that for the simpler agents shown in Figure 8.2 shows the effect of allowing cliques to change their maximum size. Instead of decreasing smoothly the E^2 value now fluctuates as cliques grow to cover several data clusters and are then split again. To limit the amount of joining and splitting that takes place, making the data more readable, we use the version of *searchForConnections* that allows cycles within the connection graph of a clique (Figure 8.1). Figure 8.8(b) was obtained by counting, for each expected data cluster, the number of points it had in common with each clique at a given point in time. First each data cluster was associated to the clique with which it had the most points in common, and the data cluster points that were not found in their corresponding clique were counted, obtaining the "lost agents" series. This series describes how many agents did not find their correct clique, but does not account for cliques that cover more then one data cluster. Thus, in the "incorrectly grouped agents" series each clique was associated with the data cluster with which it had the most points in common and points in the cliques that did not belong



(a) total squared error (b) lost and incorrectly grouped agents

Figure 8.8: Sample trial, 36 clusters of 50 point each, growth agents.

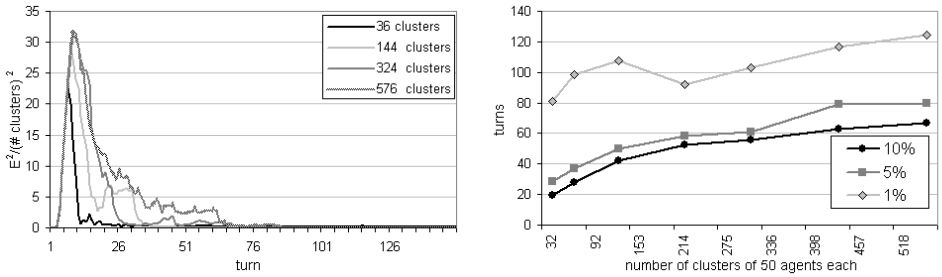
with the corresponding data cluster were counted. Note that if an agent is split off from its clique it will be counted as a "lost" agent, but if it attaches to another clique it will also be counted a second time as an "incorrectly grouped" agent. Figure 8.8(b) indicates that the fluctuations in the total squared error in 8.8(b) are caused by cliques growing too large, incorrectly joining, then readjusting to the correct size. For instance, at time 116 the value of 50 for the "incorrectly grouped agents" is the same as the cluster size, indicating that at this point there is one clique that covers two data clusters.

Figure 8.9 shows how the convergence behavior plotted in Figure 8.8 changes as we increase the size of the data sets. Figure 8.9(a) shows the total squared error over time for sample trials of data sets with between 36 and 576 clusters. Figure 8.9(b) was obtained by running 50 trials for each data set in the series and recording the turn number at which both the number of "lost agents" and "incorrectly grouped agents" first drops below 10%, 5% and 1% of the total number of agents in the data set. We then plot the average of this value over the 50 trials. Figure 8.9 indicates that the agent procedure scales less then linearly with the number of equally sized clusters in the data set. Centralized clustering algorithms have a processing cost of N or N^2 and would require a linear communication cost to gather distributed data. Thus, the agent procedure could compare quite favorably for very large distributed data sets, in spite of the fact that the communication required between agents results in it being slower when comparing the clustering of smaller data sets existing on a single machine.

8.6 Quality of Clustering

There are many data set characteristics that can affect the quality of clusterings: the shape of clusters, noise, dimensionality of data, and so forth. For this reason it is very difficult to say exactly how good the clusterings found by an algorithm are. In the absence of a standard measure of quality we show a comparison between our agent algorithm and three well-known fundamental clustering algorithms: k-means, the minimal spanning tree version of hierarchical clustering, and a simple implementation of density based clustering. We choose a data set to have characteristics that show the weaknesses of each.

Figures 8.10, 8.11, 8.12 and 8.13 show example clusterings of this data set by each algorithm. The data set consists of four areas, each intended to illustrate an aspect that can



(a) normalized E^2 (b) time to place a given percentage of agents correctly

Figure 8.9: Comparison of various sized data sets.

make clustering difficult. Area A has a row of clusters that are connected by increasingly dense bridges of additional points. An algorithm can interpret these bridges as part of the clusters, resulting in neighboring clusters being joined. Area B shows two further data features. First, it contains elongated clusters, which, because of their odd shape, are often split up. Second, the clusters increase in density from left to right, so that the clustering algorithms have to deal with data points that are separated by different distances in each cluster, and with clusters with unequal numbers of data points. Area C extends this concept further. In area B each cluster has twice the number of points as the previous one, moving from left to right. Area C accentuates this difference in density. It contains a square of 49 small clusters, containing 20 points each in the bottom left corner, and three sparse clusters produced by translating one point from each of the 49 small clusters. Finally area D illustrates the effect of noise by showing a grid of well separated clusters surrounded by increasingly more random points, from bottom to top. Like bridges, noise points can be interpreted as joining two clusters. To stress this effect the basic round clusters were generated using a Gaussian distribution from the center in place of the even distribution we used in earlier data sets.

Figure 8.10 shows the clusters found by the k-means algorithm [20], run with $k = 88$, the intended number of clusters in the data set, and random initial centroids. In the k-means algorithm a user picks the number of clusters to be found, k. The algorithm proceeds in rounds. In the first round k points are chosen as "centroids" for clusters, and all the other points are then placed in the cluster of the centroid nearest to them. In the following rounds centroids are recalculated to the be at the midpoint of each cluster and points are reallocated accordingly. The quality of the clustering is measured by its E^2 value, and rounds are repeated until this measure becomes fixed. By characterizing clusters by a single midpoint, k-means makes the assumption that clusters will have a round shape. This works well in sections A and D of the data set. It cannot however handle oblong clusters, as can be seen in section B, and even attempts to make round clusters by combining neighboring sections of two oblongs. Further, the clustering it finds is dependent on the initial selection of centroids. If two centroids are initially chosen from a single data cluster, that cluster can end up split in the final configuration. Further, since k is fixed, splitting a cluster somewhere in the data set results in two clusters being joined elsewhere. This can lead to problems when clusters have a large difference in numbers of points or densities. In section C the large sparse clusters

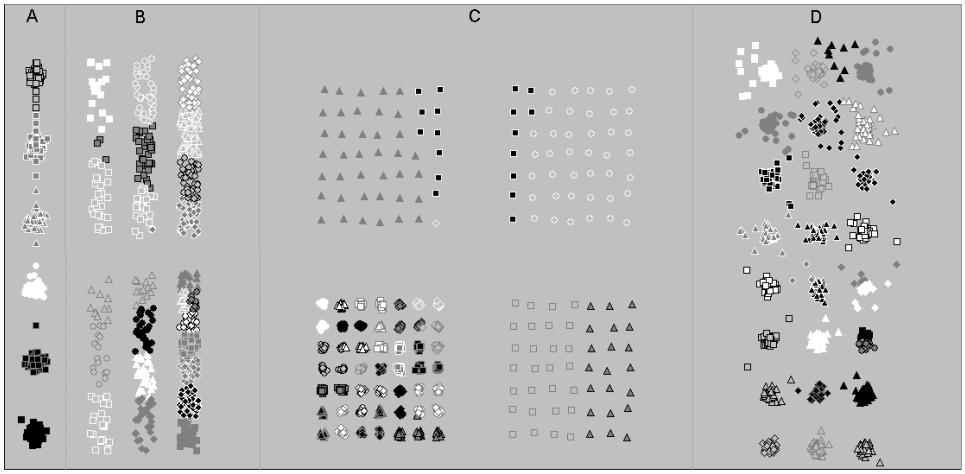


Figure 8.10: K-means clustering, k=88

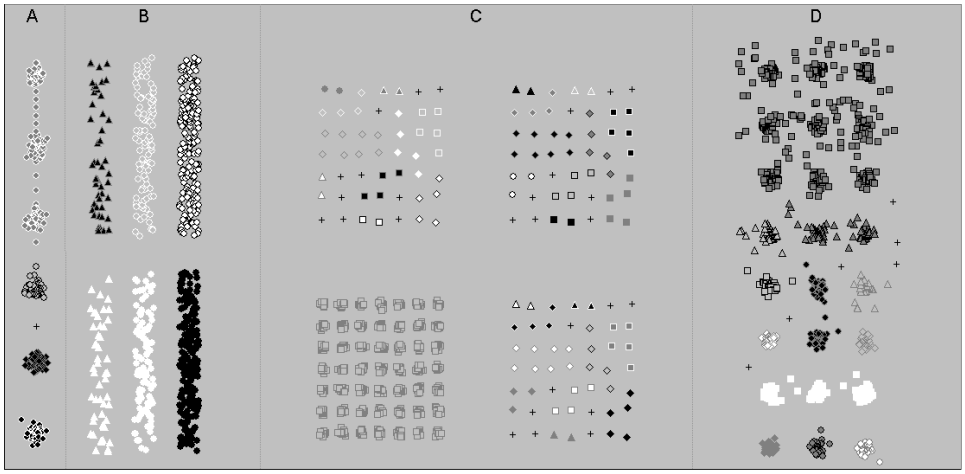


Figure 8.11: MST clustering, k=88. +'s indicate single point clusters.

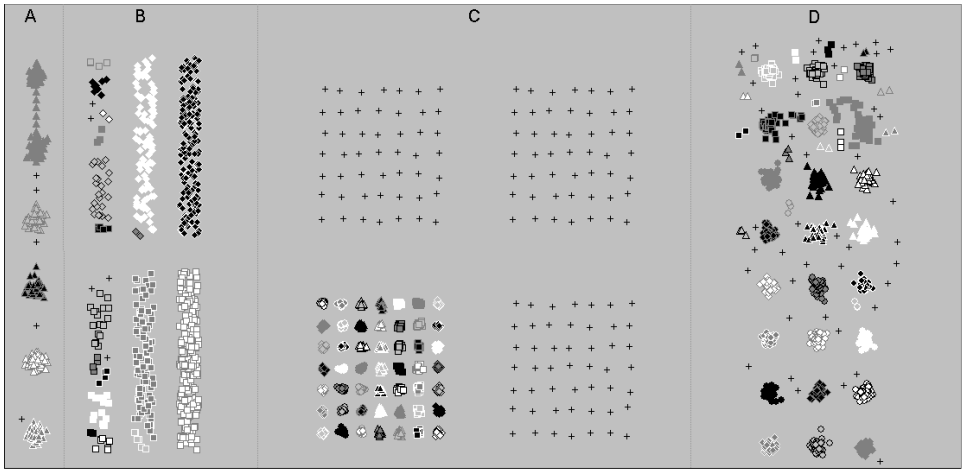


Figure 8.12: Density based clustering, d=1.5. +'s indicate single point clusters.

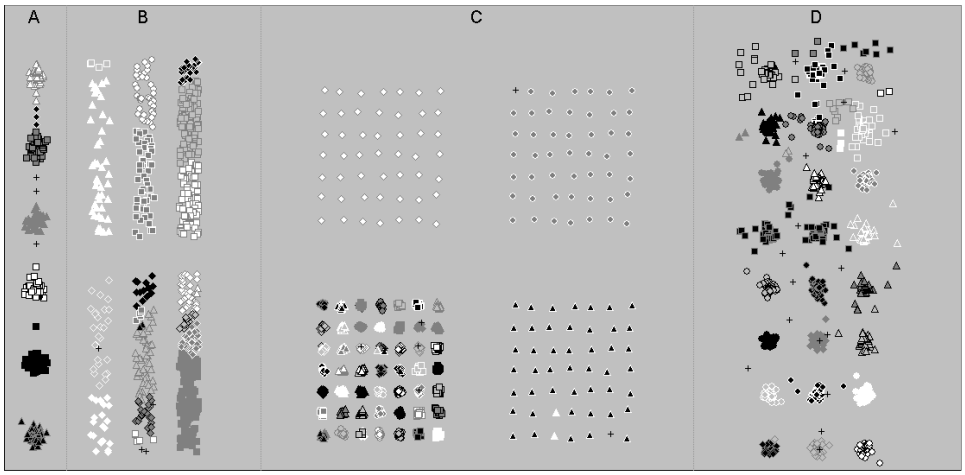


Figure 8.13: Agent clustering, γ=6. +'s indicate single agent clusters.

are split while many of the smaller dense clusters are incorrectly joined.

Hierarchical methods [15] are another widely used form of clustering. In hierarchical clustering a series of clusterings is created either by starting with single point clusters and repeatedly merging the nearest two clusters, or the other way round starting with a single cluster and repeatedly splitting it. In doing this, the assumption is made that clusters are of roughly equal density, resulting in the globally largest gap between points being an actual space between clusters. In Figure 8.11 we show the result of a basic form of hierarchical clustering, performed by building the minimal spanning tree between data set points and then breaking its longest edges. Again we show the level in the tree corresponding to 88 clusters, the number we know our data set to contain. We see in section B that this method of clustering is much better suited to oblong clusters, since it makes no assumptions about cluster shape. It however cannot deal with differing cluster densities. In sections A, C, and D it first completely divides the very sparse clusters into single elements before getting around to splitting up dense clusters that are packed together more closely. More advanced versions of hierarchical clustering include heuristics to deal with noise and bridges, however the situation illustrated by area C is fundamentally impossible for a clustering algorithm based on a global density measure to distinguish.

Density-based clustering, shown in Figure 8.12, also assumes that clusters can be parameterized by a characteristic density. In this figure we show a simple version of density based clustering in which we set a fixed global distance of $d = 1.5$ as the maximum distance between two points in the same cluster. We see this point of view avoids having to find the correct k level which is an improvement over the minimal spanning tree algorithm. It still, however, cannot simultaneously distinguish both the dense and sparse clusters in sections B and C. There is a more complex version of density-based clustering which allows different clusters to have different densities, assuming instead that clusters contain internally an even density distribution, and thus storing a density parameter separately for each clusters [52]. This algorithm takes a step towards the agent approach of describing the local characteristics of each cluster, and should be able to handle all the clusters we picture in our data set. Again though, we can always create a data set with which it will have difficulties by gradually changing the distribution of points within clusters, requiring it to become even more agent like in its ability to adapt. Density based clustering has the further weakness that it tends to follow bridges between clusters, as has occurred at the top of section A.

Figure 8.13 shows an example cluster found by our agents, using $\gamma = 6$. Since the agents make the same assumption as the density-based algorithm (that clusters are of roughly even densities) we expect it to perform in about the same way. Surprisingly, it manages to cut the dense bridge in section A, but unfortunately also splits up the oblong clusters in section B. This is due to the fact that agents only work with the best connections between points that they have found, not the best ones that exist. Since agents in the middle of a cluster have many more near neighbors then points at the edge of a cluster, or agents in a bridge, they are more likely to find the close connections that hold the cluster together. In dealing with bridges in section A, or with noise in section D this is advantageous and results in better clusterings then expected. However it produces problems for attenuated clusters. The main advantage of the adaptation that occurs in agent clustering can be seen in section C where there agents were able to correctly identify both the very sparse and closely packed dense clusters.

8.7 Conclusions

In this chapter we explored how cliques might adapt to mimic clusters contained within the underlying data set represented by the agents. We obtained some success for 2D spatial data, showing that cliques can find natural cluster boundaries, assuming that clusters are defined as dense areas of points surrounded by less dense areas. We showed that this form of clustering scales well with the number of clusters and produces clusterings that compare favorably to those created by more traditional centralized algorithms. We also found that decentralization had a great advantage in that it allowed agents to learn parameters independently for different areas in a data set. This allows the agent-clustering method to identify a wider range of clusters within a single data set than can be found using a single defining characteristic for the entire set.

We found that the most important limitation, when attempting to discover generic clusters, is that agents must have some heuristic for placing cluster boundaries. We were unable to develop an entirely parameterless boundary heuristic, due to the fact that there is no clear general definition of what constitutes a data cluster. Thus the choice of dividing a particular group of items into a single or several data clusters is always somewhat arbitrary. We explored one possible heuristic for placing cluster boundaries based on changes in the density of points within a region of the data set. This heuristic requires a single parameter, γ, to define the difference in density necessary to indicate a cluster boundary. We found that the correct value of γ for a data set is not wholly independent of cluster size. However, we showed that a single γ value could allow agents to find clusters ranging in size from 10 to 500 points. This restriction is probably reasonable for many applications. For instance, it defines a practical range for the number of replies returned by a search query.

In the case of more general data, this chapter demonstrates how difficult it is to discover abstract clusters. In many instances, especially for more complex data, clusters are unlikely to be clearly defined, or have a single parameter that can be used to choose their boundaries. Attempting to find perfect clusterings, as studied in this chapter, might therefore be more trouble than it is worth. Instead, simpler restrictions, like a maximum clique size based on available resources, could in fact be more appropriate.

Chapter 9

Summation

In the previous chapters we have shown how matchmaking and clustering can be performed among autonomous agents in a decentralized setting. This work takes a takes a step towards defining a method of decentralized search among independent entities. It also raises a number of questions for future research.

9.1 Future Directions - Search in Semi-Clustered Overlays

The simplest manner of finding items in a peer-to-peer network is to query random nodes until an appropriate reply is encountered. Organizing peers can decrease the time that such a search requires. A variety of systems have been proposed in which peers organize themselves into some form of distributed directory structure. The clusters formed by our agents can quite naturally be used in the same manner. A particular agent could be found by first searching for the correct cluster for that agent, using matching links to navigate between similar clusters, and then searching within that cluster for the agent itself. Since both the number of clusters and the size of clusters are much smaller than the entire object space, this method should improve efficiency over a direct search. This approach could also be extended by recursively clustering clusters to form a hierarchical directory.

Such solutions, however, face a problem in that the structure of the distributed directory must somehow be decided. Existing peer-to-peer systems employ three general categories of organization; distributed hash tables (DHT's), pre-existing categorizations, and discovered categories. DHT's [36] create a directory by using an ordering on objects' names. This works well for searches based on a single predetermined characteristic, but for general content-based search such a simple ordering does not exist. For some types of content, on the other hand, well defined indexes do exist. Hector and Garcia-Molina [5], for example, study the effectiveness of using a human-defined standard music categorization when locating songs in an existing peer-to-peer network. When such indexes do not exist, systems must resort to discovering categories at run time. This is done in both the traditional content based manner [48] or, in recommender type systems, based on usage patterns [3]. Interest based grouping, however, is not as simple as it first seems since, similar to text clustering, it requires a good method of choosing the significant dimensions of interest vectors.

Our matchmaking agents could make use of any of the above forms of organization by plugging in an analogous matching function. Fixed categories could be used to define cluster boundaries, as could the query radius used in recommender networks. Any such

organization, however, faces the problem that it creates a single, relatively static, structure. If the autonomy of searchers is to be maintained any such single structure is likely to be insufficient. How easily a search can be done using any directory strongly depends on how well the search criteria match up with the grouping criteria used to create the directory. In essence, a pre-defined clustering of peers could be too rigid a structure for the purposes of general search.

Considering these arguments leads us to the conclusion that our attempts to define agent heuristics for discovering cluster boundaries, in Chapter 8, may be a step in the wrong direction. While clustering can work well for data that contains obvious groupings for which such heuristics are known to exist, as we saw for 2D points, it may not lead to a solution for the broader problem of autonomous search. Rather than attempting to create a fixed clustering, a better source of criteria for cluster composition and boundaries may in fact be the actual queries being made to the system at a particular point in time. When defining a query a user usually has in mind the characteristics of the set of answers, or in other words cluster of objects, he wishes to be returned. Queries thus each contain particular clustering criteria such as cluster size, a cluster's central point, a distance metric and weight values for various data dimensions. Based on this observation we can define a system in which agents normally exist in a semi-organized state, according to whatever matching criteria they have available, and only group into more defined clusters when queries are made.

We can use our clustering agents as the basis for such a system, allowing each agent to use whatever matching function it prefers. Rather than attempting to find sensible clusters, agents could instead be limited to finding "pre-clusters" of a small fixed size, probably on the order of 5 to 10 agents. Queries could take the form of additional agents which, when added into the system, act as clique coordinators, redefining link strengths and choosing and breaking connections according to their own criteria. This could be achieved in Figure 8.1, for example, by having Cliques that represent queries use their own *linkDistance* function instead of that of the Ports. Once a query has found an appropriate cluster it can be removed from the system, allowing the agents to relax back into the semi-clustered state. Multiple simultaneous queries on different portions of the data set should not interfere with each other, while queries on the same data could either piggyback on each other or wait to execute sequentially.

If query strength functions are closely related to those the agents use, queries should be able to quickly find reply sets in such a network by grouping together the appropriate neighboring pre-clusters. If strength functions differ, the control queries have over choosing connections should serve to reinforce their view of the data, eventually evolving an appropriate cluster. The time this takes is likely to be a function of the difference between the query's strength criteria and those of the agents. It is possible that an agent uses a strength function that is completely opposite to that of a query that wishes to find it, meaning that it will never match to that query's cluster. To avoid this we could allow queries to choose connections from among all links, not just matching ones, so that random mixing could (eventually) allow such agents to be found. Taking this further we might find we need to decouple the matching and connecting properties of links, or even remove these properties all together and simply relate the length of time a link is maintained to its strength.

How well such a system would work depends on how quickly cliques form and change among our clustering agents, especially for dynamic data sets in which agents are added

and removed from the system. The timing experiments we have done in previous chapters give some indication that this process should be reasonably swift. Another concern is how well agents will deal with heterogeneous strength criteria. We have done some initial experiments on text data in which cliques developed their own word weight vectors and reassigned link strength based on each agent's distance from this central vector. We found this model produced good clusters. If find, in future work, that strength criteria can be significantly different for various agents and cliques, we may be able to create a highly flexible form of search organization in which the problem of deciding structure beforehand is greatly reduced.

9.2 When Might Decentralized Search Be Preferable to the Use of Centralized Directories?

In this thesis we have undertaken an experimental exploration of one possible model of decentralized search. The simulations performed have given us some insight into the interesting properties and limitations of this form of search. On the other hand, these abstract experiments on their own are not enough to conclude that peer-to-peer search will play a role in future computer applications, or even that the model we have chosen to study follows the most suitable approach to the problem area. One limitation of the work presented is that simulations can only give us an indication of general system behaviors. In order to understand these behaviors exactly we would have to resort to an analytical approach. However, to be feasible, such an analysis would require focusing on particular properties within a simpler model than that which can be studied through simulations. On the other hand, to determine if the peer-to-peer approach is suitable in real applications we must first study some more complex aspects of the problem, as indicated in the previous section. This requires expanding, rather than simplifying, the model. We can, however, make some speculative guesses as to what applications, if any, peer-to-peer search is most fitted

Centralized search methods are well suited to efficiently finding things that are easily and uniquely nameable, and frequently sought after. Web searches, for instance, for popular entities such as "Vrije Universiteit Amsterdam" or "Pyrrhos" are very likely to return the desired homepage. Thus web items representing things like large businesses, researchers, or major news items are all easily findable via directories. As such items are what users usually wish to find, centralized search is probably sufficient for the majority of everyday queries made directly by users.

Decentralized search, on the other hand, could be valuable when making less-popular queries for items that cannot easily be located through a directory. The use of decentralized search is limited by the fact that is is best suited to non-specific searches for items that are relatively common. Decentralized approaches are thus probably most advantageous for searches involving complex similarity functions, and thus unpredictable queries over combinations of properties which, while often present, are not worth indexing. Decentralized search can also have a technical advantage in systems that must scale to contain large numbers of items, or in systems where items change frequently. While, in general, it is possible to build large centralized servers to enable search for such data sets, not all applications are commercially viable enough to make such servers worth maintaining.

Two examples of queries which are hard to answer using a central directory are: "stores

in Amsterdam that sell items that would make good gifts to put in my Christmas crackers this year" or "recent photos of people who I knew in university". An experienced search-engine user, given sufficient time, could find answers to these queries, but doing so would require much trial and error with keywords, as well as trawling through many incorrect suggestions. Both queries, therefore, might make good candidates for decentralized searches. Many stores do list the items they currently sell in their own online catalog, and while "fits in a 5 cm diameter tube, suitable for all family members, and costs less than 2 euros" is not a usual way of indexing such catalogs, often this information is available in an item's description. Similarly, many people place their photos on the web, but as most other web users are not interested in seeing them, these photos are not generally not indexed anywhere. Further, determining if a photo's subject matches the description "people who I knew in university" requires some form of specialized similarity function. An analysis of my e-mail archive might discover that one person whose photos I would be interested in was "Matt, who was going out with Hannah, and went to work for BBN." However, ascertaining if a given photo was of this said "Matt", might require an additional query to obtain information not stored with the photo. Moreover, such photos are often private, but might be available to me if my search agent could authenticate that I was the requester.

Decentralized search systems, thus, might allow users to find things which they do not know how to describe, and, possibly, things they did not even know they wanted. In practice, such searches may not be frequently made directly by users, but they might become common in behind the scenes applications. Recommendation systems, for instance, require a method of discovering users with similar profiles. Similarly, queries to a database for a particular item might, for efficiency reasons, actually be for one of many cached copies of the item, and include requirements, unknown to the user making the query, about performance characteristics.

9.3 Other Related Research Directions

While we have addressed related work directly relevant to matchmaking and clustering within the appropriate chapters, there are a number of more general questions about peer-to-peer search that also deserve some attention. In the following sections we discuss some of the broader research questions arising from the work we have done, and indications of answers that can be found in the literature.

9.3.1 Is P2P Search Effective in the Real World?

Current interest in peer-to-peer systems is due, in part, to the success of file sharing systems for exchanging music files. This application is, however, somewhat artificial. If we consider the nature of music data we find that the set of high quality songs is fairly fixed and only grows slowly. Moreover, music is not a naturally distributed data set, song files are produced centrally, by bands conglomerated as record companies, and once created do not change. These characteristics suggest that, from the technical point of view, music is well suited for distribution from central servers. Thus the popularity of file sharing networks could be seen as a temporary phenomenon, due to the absence of music servers and the high cost and difficulty of obtaining music via the existing CD distribution method.

Future applications of peer-to-peer file sharing are more likely to focus on data that is produced and changed in a more distributed manner. Whereas music sharing is dominated by a small number of very popular files, more distributed data is likely to be valuable because of it heterogeneity. This, however, results in a problem for peer-to-peer systems, which are far better at finding popular items than rare ones.

This raises the question of how broad the range of situations is in which peer-to-peer networks are able to locate items, especially rare items. We faced a similar problem when we found that the probability of two randomly chosen ports matching must be above a certain level before cliques would form among our agents. One way of combating the problem of finding rare items is to attempt to organize the system. As we have found in this work, however, such organizations are not easy to build and are probably not always possible. Therefore, it is also interesting to know the limits of laxer techniques that rely only on random searching.

In [3] the estimation is made that Gnutella-like flooding searches in random networks, with a typical query horizon, are roughly equivalent to querying 1000 random member peers. For the web proxy data sets studied in that paper the authors find that between 80% and 90% of queries can be answered by such random probes, but that only 1.3% to 14% of queries for items that are located on at most .01% of nodes are successful. It is hard to tell how well our agents would handle such scarce data items. It is probably reasonable to make the same approximation that, when searching, our matchmaking agents simply query a number of random other agents. Estimating from Figure 3.10 that the number of categories supported doubles with each port added, about 9 ports per agent would be needed to support the 10,000 categories that occur in the data set used in [3]. This is, however, for an even distribution of items. Since the data set used also contains very popular items it is likely that these would drive the matchmaking process and the number of ports per agent needed would in fact be lower. The required to find rare items, on the other hand, would need to be determined through further experimentation. This time could probably be improved by replicating rarer items [4]. A good replication strategy can make rare items easier to find, while making popular items only slightly harder to locate. On the other hand, replication cannot change the fundamental problem that search becomes more difficult as the number of different items increases.

9.3.2 What Form of P2P Organization Best Supports Search?

In this thesis we study how search can be performed within a peer-to-peer network in which both items to be found and queries are represented by network nodes. We consider a single network model, with a number of fixed properties. In this model groups of nodes are explicit, and coordination within groups is assumed. We allow nodes to only cooperate within their clique, thus keeping open the possibility that a node cannot necessarily communicate with all other nodes. Nodes maintain no information about each other's data, and thus we do not allow for middle men in the query process. This allows agents to each have different measures of similarity. We considered a dynamic network, with no predefined restrictions on organization. We further considered incoming communications to a node to be about as costly as outgoing communications, and thus created symmetric links between nodes. These properties derive from the fact that we are particularly interested in search between autonomous agents. Our design also reflects the fact that we speculate that the process of

choosing a reply set for a query is similar to the process of clustering, and in particular to the process of discovering cluster boundaries. Thus, in this work, we were most interested in the properties of the groups of nodes formed within our network, rather than in the properties of the network graph. Other forms of network can also be used to study peer-to-peer search. It is an open question what peer-to-peer network model would actually be most effective for the search problem.

What Form of Network Graph is Should Be Used to Implement Search?

The complexity of peer-to-peer nodes has a large influence on the most appropriate network with which to connect them. Yu *et al.* [54], for instance, study advanced agents which maintain models of their neighbor's expertise and sociability. Queries are performed by passing requests from node to node. Through their models agents form a map of their neighborhood which they use to guide the routing of queries. Since creating these maps is costly, such networks work best when they do not change rapidly. Additionally, maintaining maps means that maintaining an outgoing link is far more expensive than having an incoming link. The authors therefore choose to make links unidirectional. In this model nodes do not join into groups, except implicity by way of the structure of the link network. In fact, the authors find that search improves the less nodes group together. This is because less grouping results in a shorter the average path length between any two agents, and thus a shorter expected distance that a query must travel to find a particular node.

Voulgaris and Van Steen [49], on the other hand, study extremely simple nodes, connected by a highly dynamic network of links. The aim in this work is to form links between semantically similar neighbors, using an epidemic protocol to discover high-quality links. The network is modified by a gossip protocol through which nodes exchange neighbors. Nodes' data is assumed to be simple enough that nodes can store the data item of each of their neighbors. Node can thus pass on links which they calculate to be most similar to another node, speeding the rate at which good-links are found. One of the aims of this model to change links quickly, thus creating a highly adaptive, robust, network. Unidirectional links are therefore appropriate since modifying them requires no agreement between the nodes being linked. In fact, the gossip protocol does not involve any explicit coordination between nodes. Instead it relies on implicit coordination derived from fact that all nodes use the same similarity function. There is also no coordinating behavior that stops nodes all choosing the same neighbors. Therefore, the model requires two protocol layers, one to maintain a source of random links and a second to form proximity the overlay. Nodes form implicit, rather than explicit, groups.

Are P2P Organizations Maintainable?

In this thesis we have studied systems with fixed sets of agents. One of the main advantages of decentralized systems is that they can easily deal with changing data, and thus also changing sets of nodes. In real applications systems must therefore deal with nodes being added and removed, and perhaps more importantly, nodes failing. When connectivity is not maintained centrally, however, over time a changing set of nodes can easily result in network partitions. Given only a view of their local area, nodes are unable to detect, or

prevent, such divisions. The manner in which nodes are added and removed from a system must therefore be carefully designed to maintain global connectivity.

Distributed hash tables [36] are one of the more thoroughly studied forms of organized peer-to-peer networks. As Ratnasamy *et al.* [39] indicate, however, the effect of nodes failing in such systems has yet to be studied in depth. Tapestry [56] is designed with a dynamic node population in mind. In this paper, the authors show how nodes can be dynamically inserted an removed from a distributed hash table, and demonstrate that the performance of their routing network degrades gracefully in the presence of node failures.

Jelasity and Van Steen [16] study the effect of failure in another form of peer-to-peer network in which links are modified by nodes gossiping to learn about new neighbors. In their protocol nodes continually exchange neighbor lists, and select new neighbors as they hear about them, replacing old ones. They find this protocol effectively handles adding and removing nodes and can also withstand failures of large numbers of nodes. They do, however, observe that the system can, on rare occasions, spontaneously partition, even without nodes failing.

Partitioning can also occur when the connections between nodes are modified. Many peer-to-peer systems use flooding queries over a fairly static network. In this approach a nodes neighbors are queried for an item. If a neighbor does not have the item being sought, it passes the request on to its neighbors. In this work, on the other hand, we follow the philosophy that if a neighbor does not have the required item, it is time to change neighbors. Changing neighbors avoids middle men, and thus problems like creating reasons why neighbors would forward queries, or making translations over chains of nodes. Changing neighbors, however, also results in a highly dynamic network which is more prone to partitioning. Thus this form of query may prove to be more suited to requests for longer term collaborations than to frequent simple searches that only return small amounts of data. In the experiments that we have run so far we have yet to observe problems with partitioning. Though we did not specifically study network connectivity, the fact that clusters were found correctly in large data sets indicates that the agents do not get separated into different partitions. Partitioning could, however, become more of an issue in very large or long running systems, or in systems where the links to maintain are partly determined by location. On the other hand, if search queries can have many possible answers, as long as partitions are not too small, searches will still succeed in a partitioned network. Difficulties will only arise when searching for items, which, due to rarity or some other effect, do not occur within a particular partition.

9.3.3 What Metrics Are Best for Determining Similarity?

One of the main problems encountered when organizing high dimensional data is choosing a method by which to measure which points are most similar to which other ones. Though humans often perceive patterns in complex data, often such similarities are unmeasurable by computers. Moreover, for complex data, especially when user opinion is involved, it is unlikely that definitive similarity metrics exist. Thus, straightforward, measurable metrics that simply work well enough, are often more usable than complex ones.

Intuitively, for well-know applications areas it should be possible to design custom similarity metrics. Crespo and Garcia-Molina [5] consider using an existing classification hierarchy to organize nodes from a trace of a Napster music sharing network, according to the

song files they contain. They use an expert human classification for data that is relatively well understood and not extremely complex. It thus seems fair to assume that the classification they use is of close to the best quality that we could hope to have for an application area. Yet, even with this high quality information, they find many problems in attempting to create an organized search structure. First, they needed to divide nodes into clusters of similar nodes. Their classification hierarchy divided songs into 26 general style categories, and 255 more specific substyles. They allowed nodes to be in more than one cluster. Based on the classification of the individual files they contained, however, only 24% of nodes clearly belonged to only one of the 26 general clusters. They thus needed to introduce a parameter that set the percentage of files a node must have from a particular cluster to become a member of that cluster. More importantly, they found that the style clusters did not evenly divide the data set, and thus would not always significantly restrict the search space. For the 26 general clusters, 14 contained less than 22% of the nodes, but one cluster contained almost all of the nodes. For the 255 substyle clusters they got better results, 222 of them contained less than 22% of the nodes, but they still found one large substyle that contained 57% of the nodes. A second problem they faced was that files on nodes needed to be classified in order to determine the nodes' clusters and queries also needed to be classified in order to choose which cluster they should be sent to. They used an automated classifier that had a list of known song classifications to identify song files when sorting the nodes. This classifier, however, could only classify 75% of files correctly. They used human choice to classify queries. Humans were, however, only able to assign 14% of their queries to a substyle, and thus most of the queries were directed to the less restrictive style clusters. Moreover, they noted that the queries that were easily classifiable where those for popular files, which due to their high replication could be found quickly simply using random search. We must therefore consider that creating a good content-based similarity metric, even for a restricted application area, is not actually straightforward.

Recommender networks follow the principle that human interest, as measured by usage patterns, can be used to determine similarity. Cohen *et al.* [3], for example, study "associate overlays" in which replicas of data items are distributed over the network nodes, and nodes are placed in the same group if they contain a particular item of data. In particular they study web-proxy log data sets in which nodes are users and items are hostnames those users choose to access. These data items can be seen as dimensions in an interest vector, and groups as clusters each with a single distance metric, based on a chosen dimension for the cluster, which is 1 between two nodes that contain that dimension and 0 otherwise. Unlike the agents we studied, nodes can belong to multiple clusters. Cohen *et al.* test the principle that the items people choose to search for are not independent, but that if two nodes contain one item in common they are then more likely to contain other items in common. Accordingly, when a search is made by a node it should first be made among members of groups to which that node belongs. They show that for their web proxy data 40% of items are replicated on less then 1% of the nodes, and that such group based search does indeed improve the chance of finding these rare items within a certain number of probes to other nodes.

Sripanidkulchai *et al.* [46] study a similar type of system in which successful replies to queries are used to build "shortcuts" in a Gnutella network. The idea behind this is that each peer chooses a small number (10) of shortcuts, to other peers who have returned responses

to previous queries. These shortcuts are then used when making new queries, sequentially in order of their overall success rate at answering queries. Experiments were done using data from three web traces, a Kazaa trace, and a Gnutella trace. The authors found that between 53% and 90% of the time queries were answerable though shortcuts, and that queries needed to be made to an average of 1.5 peers before finding a reply. These results indicate a high level of "interest-based locality", that is, that a peer that answered one query from a user is much more likely to be able to answer an second query from that user. The composition of these interest groups, however, is not clear. Such results could be obtained if small groups of peers are very similar to each other, indicating a system in which can be easily divided into clusters. They could, however, also be the result of having a small number of peers that contain large amounts of content, and are thus good candidates to form shortcuts to. The paper's authors report that on closer inspection of one shortcut graph they found a small number of well connected components which had a small-world type structure. This indicates that interest groups may in fact be quite large, though the graph they considered only had about 1000 nodes and therefore may not give a complete picture.

Choosing similarity metrics is thus likely to become a key issue in designing organized peer-to-peer systems. Similarity as observed by human experts does not necessarily create an optimal organization for the tasks we wish computers to accomplish. Recommender systems also reflect human opinion and risk users adapting to the system assumption that they should have the same tastes as their neighbors rather than the computer system adapting to users' true preferences. Ultimately, peer-to-peer systems will have to face the issue that many users could result in many simultaneous differing views of the system data. Combining these views to form a global organization is not an easy issue. Aberer *et al.* [1] for instance, studied how peer-to-peer translations among many local data schemas can be used by nodes to come to a consensus on a single global schema. They face a problem, however, with the chaining of transactions, forced by the peer-to-peer model. The global schema used between two nodes can only contain concepts that are correctly translatable among all of the nodes in the path between them. In general, we may find that systems need to be designed to operate according to many views, rather than attempting to create consensus.

9.4 Final Conclusions

This thesis has addressed the research question:

> Is it possible to effectively and efficiently match pairs, or group together sets of items, based on their content, without creating a centralized directory or placement scheme?

We studied a model in which data items are represented by autonomous agents joined in a peer-to-peer network. We have shown, through simulations using abstract data sets, that we can design agents that quickly find high quality solutions to matchmaking and clustering problems, under certain conditions. We showed that the peer-to-peer approach resulted in systems that could accommodate large numbers of agents, with the time needed to find matches or clusters increasing less than linearly with system size. We have further demonstrated two applications, a double auction and text clustering, indicating that the conditions under which such agent algorithms can be used are not overly restrictive.

We found that the basic behavior of our model depends on the probability that two random agents in the system match. If this probability is sufficiently large the peer-to-peer system will yield a good solution for matchmaking and clustering problems. When this probability is too small no solutions are found, but, in many cases we can adapt the system to obtain the desired behavior. This can be done by increasing the number of neighbors of each agent, or by enhancing the flexibility with which agents accept others as matches. We found that simple learning procedures were especially valuable in producing this flexibility. We also studied a learning procedure that allowed agents to adapt their clustering behavior to fit with a wide range of data sets. In particular we showed how agents could learn the expected density and size of clusters at different areas within a data set, giving them an advantage over centralized clustering techniques which stipulate a single global value for cluster characteristics.

Our motivation for this work was to determine if decentralized search could have advantages over the use of centralized directories. For both matchmaking and clustering we showed improvements in scalability over centralized algorithms. We believe ultimately, however, that the real advantage of decentralized methods will come from their fine grain distribution of control among searchers and the owners of data to be search. Such a distribution allows data to be independently updated without having to inform other parties, and allows queries and replies to be tailored to data suppliers or requesters.

Bibliography

[1] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth, The Chatty Web: Emergent Semantics Through Gossiping. In *International World Wide Web Conference (WWW)*, (2003).

[2] D. Cliff and J. Bruten, Zero is not enough: On the Lower Limit of Agent Intelligence for Continuous Double Auction Markets. Technical Report HPL-97-141, Hewlett-Packard Laboratories (1997).

[3] E. Cohen, A. Fiat, and H. Kaplan, Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. In *Proceedings of the IEEE INFOCOM'03 Conference*, (2003).

[4] E. Cohen and S. Shenker, Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proceedings of the ACM SIGCOMM'02 Conference*, 2002.

[5] A. Crespo and H. Garcia-Molina, Semantic Overlay Networks for P2P Systems. Technical Report, Standord University, 2003.

[6] K. Decker, K. Sycara, and M. Williamson, Middle-Agents for the Internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, (1997), 578–583.

[7] T. Eiter, D. Veit, J. Müller and M. Schneider. Matchmaking for Structured Objects. In. *Proceedings of the Thrid International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2001).* , Springer Lecture Notes in Computer Science series vol. no. 2114, (2001), 186–194.

[8] M. Ester, H. Kriegel, J. Sander, and X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Nose. In *Second International Conference on Knowledge Discovery and Data Mining*, (1996), 226–231.

[9] C. Faloutsos and D. Oard, A Survey of Information Retrieval and Filtering Methods. Technical Report CS-TR3514, Computer Science Department, University of Maryland, 1995.

[10] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, English ed., Addison-Wesley, Edinburgh, (1999).

[11] L. Forner, Yenta: A Multi-Agent, Referral-Based Matchmaking System, In *Proc. of the 1st Int. Conference on Autonomous Agents*, (1997), 301–307.

[12] D. Gode and S. Sunder, Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality. *The Journal of Political Economy* **101**(1), (1993), 119–137.

[13] S. Guha, R. Rastogi, and K. Shim, CURE: An efficient clustering algorithm for large databases. *Information System Journal* **26**(1), (2001), 35–58.

[14] A. Jain and R. Dubes, *Algorithms for Clustering Data*. Pentice-Hall advanced reference series. Prentice-Hall, (1988).

[15] A. Jain, M. Murty, and P. Flynn, Data Clustering: A Review. *ACM Computing Surveys* **31**(3),(1999), 264–322.

[16] M. Jelasity and M. van Steen. Large-scale newscast computing on the Internet. Technical Report IR-503, Vrije Universiteit Amsterdam, (2002).

[17] S. Jha, P. Chalasani, O. Shehory, and K. Sycara, A Formal Treatment of Distributed Matchmaking. In *Proc. of the 2nd Int. Conference on Autonomous Agents*, (1998), 457-458.

[18] N. Jennings, K. Sycara, and M. Wooldridge, A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, **1**(1), (1998), 7–38.

[19] D. Judd, P. McKinley, and A. Jain, Large-Scale Parallel Data Clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**(8), (1998), 871–876.

[20] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: an Introduction to Cluster Analysis*, John Wiley and Sons (1990).

[21] M. Klusch and A. Gerber, Dynamic Coalition Formation Among Rational Agents. *IEEE Intelligent Systems* **17**(3), (2002), 42–47.

[22] D. Kuokka and L. Harada, Matchmaking for Information Agents. In *Proc. of the 14th Int. Joint Conference on Artificial Intelligence*, (1995), 672–678.

[23] B. LeBaron, Agent Based Computational Finance: Suggested Readings and Early Research. *Journal of Economic Dynamics and Control* **24**(5-7), (2000), 679–702.

[24] S. Mullender and P. Vitányi, Distributed Match-Making. *Algorithmica* **3**, (1988), 367–391.

[25] R. Ng and J. Han, Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proceedings of the 20th VLDB Conference*, (1994), 144–155.

[26] E. Ogston, B. Overeinder, M. Van Steen, and F. Brazier, A Method for Decentralized Clustering in Large Multi-Agent Systems. In *Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, (2003), 789–796.

[27] E. Ogston, B. Overeinder, M. Van Steen, and F. Brazier, Group Formation Among Peer-to-Peer Agents: Learning Group Characteristics. In *Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, Springer Lecture Notes in Computer Science series vol. no. 2872, (2004), 59–70.

[28] E. Ogston, M. van Steen and F. Brazier, Group Formation Among Decentralized Autonomous Agents. *Journal of Applied Artificial Intelligence* **1**(9-10), (2004), 953–970.

[29] E. Ogston, M. van Steen, F. Brazier, and B. Overeider, Data Clustering by Large Scale Adaptive Agent Systems. Preprint.

[30] E. Ogston and S. Vassiliadis, Matchmaking Among Minimal Agents Without a Facilitator. In *Proc. 5th International Conference on Autonomous Agents*, (2001), 608–615.

[31] E. Ogston and S. Vassiliadis, Local Distributed Agent matchmaking. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, (2001), 67–79.

[32] E. Ogston and S. Vassiliadis, Unstructured Agent Matchmaking: Experiments in Timing and Fuzzy Matching. In *Models, Languages and Applications Special Track of the 17th ACM Symposium on Applied Computing*, (2002), 300–305.

[33] E. Ogston and S. Vassiliadis, A Peer-to-peer agent auction. In *First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, (2002), 151–159.

[34] C. Olson, Parallel Algorithms for Hierarchical Clustering. *Parallel Computing* **21**, (1995), 1313–1325.

[35] A. Ouksel and I. Ahmed, Ontologies are not the Panacea in Data Integration: A Flexible Coordinator to Mediate Context Construction . *Distributed and Parallel Databases*, **7**(1), (1999), 7–35.

[36] C. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems* **32**, (1999), 241–180.

[37] C. Preist and M. van Tol, Adaptive Agents in a Persistent Shout Double Auction. In *Proceedings of the 1st International Conference on the Internet, Computing and Economics. ACM Press*, (1998), 11–17.

[38] L. Rasmusson and S. Janson, Agents, Self-Interest and Electronic Markets. *The Knowledge Engineering Review* **14**(2), (1999), 143–150.

[39] S. Ratnasamy, S. Shenker, and I. Stoica. Routing Algorithms for DHTs: Some Open Questions. In *The First International Workshop on Peer-to-peer Systems (IPTPS'02)* (2002).

[40] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGray-Hill, 1983.

[41] T. Sandholm and V. Lesser, Coalition Formation among Bounded Rational Agents. In *14th International Joint Conference on Articial Intelligence*, (1995), 662–669.

[42] O. Shehory and S. Kraus, Task Allocation via Coalition Formation among Autonomous Agents. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, (1995), 655–661.

[43] V. Smith, An Experimental Study of Competitive Market Behavior. *The Journal of Political Economy*, **70**(2), (1962), 111–137.

[44] O. Shehory, A Scalable Agent Location Mechanism. In *Lecture Notes in Artificial Intelligence, Intelligent Agents VI*, M. Wooldridge and Y. Lesperance (Eds.), (1999), 162–17.

[45] O. Shehory and S. Kraus, Methods for Task Allocation via Agent Coalition Formation. *Artificial Intelligence*, **101**(1-2), (1998), 165–200.

[46] K. Sripanidkulchai, B. Maggs, and H. Zhang, Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In *Proceedings of the 22nd Annula Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, (2003).

[47] K. Sycara, S. Widoff, M. Klusch, and J. Lu, LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems* **5**, (2002), 173–203.

[48] C. Tang, Z. Xu, and S. Dwarkada, Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Commuications* (2003).

[49] S.Voulgaris and M. van Steen. Epidemic-style Management of Semantic Overlays for Content-Based Searching. Technical Report IR-CS-011, Vrije Universiteit Amsterdam, (2004).

[50] N. Vulkan and N. Jennings, Efficient Mechanisms for the Supply of Services in Multi-Agent Environments. *International Journal of Decision Support Systems*, **28**(1-2), (2000), 5–19.

[51] M. Wellman, E. Walsh, P. Wurman, and J. MacKie-Mason, Auction Protocols for Decentralized Scheduling. *Games and Economic Behavior* **35**(1-2), (2001), 271–303.

[52] X. Xu, M. Ester, H. Kriegel, and J. Sander, A Distrbution-Based Clustering Algorithm for Mining in Large Spatial Databases. In *Proceedings of the 14th International Conference on Data Engineering*, (1998).

[53] P. Yolum and M. Singh, Dynamic Communities in Referral Networks *Web Intelligence and Agent Systems* **1**(2), (2003), 105–116.

[54] B. Yu, M. Venkatraman, and M. Singh, An Adaptive Social Network for Information Access: Architecture and Experimental Results. *Applied Artificial Intelligence* **17**(1), (2003), 21–38.

[55] T. Zhang, R. Ramakrishnan, and M. Livny, BIRCH:A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery* **1**(2), (1997), 141–182.

[56] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, (2001).

Samenvatting

Koppelen en clusteren met agenten: een gedecentraliseerde aanpak voor zoekproblemen

Inhoudsgericht zoeken is het proces van het vinden van items met bepaalde specifieke karakteristieken, in tegenstelling tot het zoeken op basis van een label of naam. In essentie sluit het in zich het individueel bekijken van ieder item. Als het aantal items groeit, dan wordt inhoudsgericht zoeken snel problematisch. Als daarnaast de items ver uit elkaar liggen in het (computer) netwerk, dan zullen de kosten voor het individueel bekijken van ieder item, een uitputtende zoekprocedure onmogelijk maken.

Inhoudsgericht zoeken in een gedistribueerd bestand gebruikt daarom vaak een samenvoegmethode. Het samenvoegen van items in een gedistribueerd bestand kan op twee manieren worden gedaan. De eerste methode is alle items in het netwerk te kopiëren in een centrale catalogus. In het algemeen vereist dit een enorme hoeveelheid geheugenruimte en een intensieve procedure om de kopieën consistent te houden. In dit geval kopieert men daarom meestal niet de gehele inhoud van een item, maar slechts een samenvatting. Deze samenvattingen kunnen vervolgens geïndiceerd worden om het zoeken verder te vergemakkelijken. In plaats van een centrale catalogus kan men de items organiseren volgens een vast plaatsingsschema, zodat men via de (virtuele) posities ieder item eenvoudig kan lokaliseren.

De ”Gele Gids” gebruikt de eerste methode en helpt gebruikers bedrijven te vinden door een catalogus of gids te raadplegen. Het ”Dewey Decimal System” werkt volgens de tweede methode en laat gebruikers boeken zoeken in een bibliotheek volgens een vast schema. In feite kunnen gebruikers die op zoek zijn naar wetenschappelijke werk direct naar de ”500” planken gaan in de kennis dat bijvoorbeeld paleobotanie te vinden is onder ”561”.

Centrale catalogussen worden veel gebruikt in computersystemen, maar zijn onderhevig aan een schalingsprobleem, zowel in het aantal items dat verwerkt en opgeslaan kan worden, als in de hoeveelheid werk dat nodig is om de catalogus up-to-date te houden. De beide methoden hebben als bijkomend nadeel dat bij het ontwerp van het plaatsingsschema en bij het opstellen van de samenvattingen speculaties gemaakt moeten worden over de specifieke karakteristieken van ieder item die belangrijk zullen zijn voor alle toekomstige zoekopdrachten.

Het centrale probleem in dit onderzoeksgebied vraagt dus om een methode om inhoudsgericht te zoeken naar items in een gedistribueerd bestand, waarbij de kosten van het on-

derhoud en de speculatieve aspecten worden vermeden. In dit proefschrift richten we ons op de volgende twee fundamentele deelproblemen. Is het mogelijk effectief en efficiënt items te koppelen of items te clusteren op basis van inhoud, zonder een centrale catalogus of plaatsingsschema te creëren?

Door simulaties met abstracte datacollecties hebben we aangetoond dat een zogenaamd peer-to-peer systeem van agenten ontwikkeld kan worden dat, onder zekere voorwaarden, snel een oplossing vindt voor het koppel- en clusterprobleem. In dit systeem representeert elke agent een item uit de datacollectie. Deze simulaties tonen aan dat een dergelijk systeem uit een groot aantal agenten kan bestaan en dat de tijd om een oplossing te vinden lineair groeit met de grootte van het peer-to-peer systeem.

Om de praktische toepassingen van het peer-to-peer systeem te illustreren hebben we het gebruikt in twee concrete problemen: het clusteren van teksten en in de zogenaamde ”double auction”. Door middel van verscheidene simulaties hebben we verder onderzocht voor welke problemen het peer-to-peer systeem gebruikt kan worden. We hebben gevonden dat het succes van het peer-to-peer systeem afhankelijk is van de kans dat twee onafhankelijk gekozen agenten in het systeem uiteindelijk gekoppeld kunnen worden. Als de kans voldoende groot is, zal het peer-to-peer systeem een oplossing vinden voor het koppel- of clusterprobleem. Als de kans te klein is, kunnen we echter het systeem aanpassen. Inderdaad, door het aantal buren van iedere agent in het systeem te vergroten en de flexibiliteit van iedere agent om aan andere agenten gekoppeld te worden te vergroten, kan het peer-to-peer systeem in veel gevallen gebruikt worden, zelfs als de kans te klein is. Een eenvoudige leerprocedure voor de agenten is bijzonder bruikbaar voor het vergroten van de flexibiliteit.

In het flexibelere peer-to-peer systeem zijn de agenten beter in staat hun koppelingsgedrag aan te passen, zodat het toepasbaar is op een groter aantal gedistribueerde datacollecties. In het bijzonder hebben we laten zien dat de flexibelere agenten de verwachte dichtheid van de clusters op verschillende plaatsen in het gedistribueerde systeem kunnen leren. Deze eigenschap van de agenten maakt het peer-to-peer systeem breder inzetbaar dan de gedecentraliseerde methoden waarin bepaalde parameters, zoals de grootte en vorm van de clusters, van te voren gespecificeerd moeten worden.

In zowel het koppel- als clusterprobleem hebben we laten zien dat het peer-to-peer systeem betere schalingseigenschappen heeft dan de centrale methoden. Een bijkomend voordeel van de gedecentraliseerde methode is dat items up-to-date gebracht kunnen worden zonder andere te informeren en dat vragen en antwoorden in de zoekopdracht aangepast kunnen worden aan de specifieke wensen van de zoeker of bezitter van items. We geloven echter dat uiteindelijk het echte voordeel van de gedistribueerde methode zal voortkomen uit de individuele controle van de zoekers en bezitters van items.